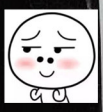
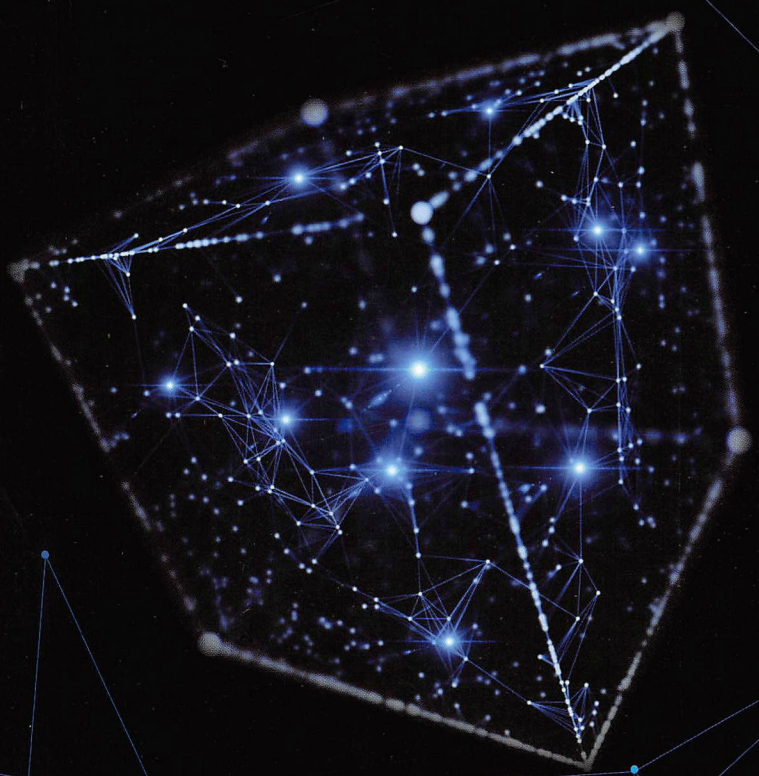
Broadview®
www.broadview.com.cn



Spark SQL 内核剖析

朱锋 张韶全 黄明 著

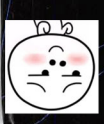


中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

朱锋，博士毕业于中科院软件所，研究方向为分布式计算与软件工程。长期关注数据分析、数据库技术和大数据相关系统，并积极参与开源社区贡献。2017年加入腾讯，负责Spark SQL相关平台的开发、优化和维护工作，在SQL-on-Hadoop方面积累了丰富的经验。

张韶全，香港中文大学博士，博士期间的研究方向为系统最优分布式算法。曾任香港应用研究院研究员、联想香港研发中心高级研究员。现任腾讯大数据平台高级研发工程师，负责腾讯大数据SQL平台的建设与研发，平台规模达到上万台服务器，百万级别业务量，PB级日数据计算量，支撑着腾讯全公司的数据分析业务。拥有多年互联网公司一线大数据平台设计与研发经验，旨在传播大数据技术和实践经验，使其在不同行业落地生根。

黄明，腾讯T4专家，Spark中国区早期研究者和布道者之一。



Spark SQL 内核剖析

朱锋 张韶全 黄明 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Spark SQL 是 Spark 技术体系中较有影响力的应用（Killer application），也是 SQL-on-Hadoop 解决方案中举足轻重的产品。本书由 11 章构成，从源码层面深入介绍 Spark SQL 内部实现机制，以及在实际业务场景中的开发实践，其中包括 SQL 编译实现、逻辑计划的生成与优化、物理计划的生成与优化、Aggregation 算子和 Join 算子的实现与执行、Tungsten 优化技术、生产环境中的一些改造优化经验等。

本书不属于入门级教程，需要读者对基本概念有一定的了解。在企业中任职的系统架构师和软件开发人员，以及对大数据、分布式计算和数据库系统实现感兴趣的研究人员，均适合阅读本书。

图书在版编目（CIP）数据
Spark SQL 内核剖析 / 朱锋, 张韶全, 黄明著. —北京：电子工业出版社，2018.7
ISBN 978-7-121-34314-8
I. ①S… II. ①朱… ②张… ③黄… III. ①数据处理软件 IV. ①TP274
中国版本图书馆 CIP 数据核字 (2018) 第 111300 号

策划编辑：张春雨
责任编辑：牛　勇
印　　刷：三河市君旺印务有限公司
装　　订：三河市君旺印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　　印张：18　　字数：390 千字
版　　次：2018 年 7 月第 1 版
印　　次：2018 年 7 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

推荐序 1

互联网技术经过几十年的发展已经渗透到人们生活的方方面面，从云计算、大数据到如今如火如荼的人工智能和区块链，相信无论是圈内人还是圈外人，对这些名词都耳熟能详了。仔细一算，“大数据”这个概念的出现已经有十多年了，背后催生的技术可以说是百花齐放、百家争鸣。

2009 年年初，腾讯从传统的数据仓库转向基于 Hadoop 架构的大数据平台，至今将近 10 年，历经了 3 代跨越式的发展：2009—2011 年是以 Hadoop 为基础的离线计算时代，2012—2014 年是以 Spark 和 Storm 为引擎的实时计算时代，2015 年至今是以腾讯自研的高性能机器学习平台 Angel 为核心的智能学习时代。从最简单的统计报表的计算，到万亿特征维度的算法训练，从结构化数据到图片、语音、文本等非结构化数据，腾讯一直用前沿技术来挖掘大数据背后的价值。

如今，腾讯大数据集群规模达到几万台服务器，存储数据量有几百 PB，每天有几十 PB 的计算量，支撑着腾讯包括微信、QQ、游戏、广告、支付、视频、音乐等关键业务，助力腾讯业务发展，服务着十亿级别的用户。正是历经了腾讯数以亿计的海量数据的锤炼，让腾讯大数据平台得到快速的发展，其技术在业内处于领先水平。

腾讯大数据起源于网络社区，并一直积极参与网络社区的建设。2014 年，腾讯大数据平台（TDW）的核心组件进行开源，我们在 Hadoop、Spark、Docker、Ceph、HBase、Kubernetes、Kafka、Storm、Flink、PostgreSQL 等众多社区项目上积极“反哺”社区。2017 年 6 月，我们在 GitHub 上把腾讯大数据第三代的高性能分布式机器学习平台 Angel 进行了开源，吸引了海内外众多知名企业用户，并于 2018 年 3 月贡献给 Linux 深度学习基金会（LF Deep Learning）。

除代码层面的开源外，近年来，腾讯也把大数据能力开放给传统企业，我们服务了政务民生、金融、交通、零售、教育、工业等各行各业的用户，旨在让没有大数据人才的企业也能具备使用大数据的能力。我们乐于把腾讯积累了十年的大数据技术和运维经验对外分享、对外输出，本书也可以看作是腾讯大数据技术开放的一部分。

本书的内容最初是腾讯内部为进行 Spark SQL 开发而整理的技术文档，最后剥离出通用的部分集结成册。从数据的维度来看，无论是单机还是分布式环境，SQL 对用户来说都是非常重要的。Spark SQL 作为腾讯大数据平台中最基础的部分，支撑全公司的数据分析业务。因此，书

中的内容并非是针对 Spark SQL 技术的空谈，而是立足于腾讯大数据平台的大量实践经验。

本书的几位作者正是工作在腾讯大数据一线的工程师和技术专家，在日均百万级别的 SQL 业务处理和优化中积累了丰富的经验。综观全书，条理非常清晰，读者既能在高度上知晓来龙去脉和他山之石，又能在深度上体会源码级别的技术点剖析。同时，书中结合实践展示了一些通用案例，避免读者陷入到代码的汪洋大海中。

于我个人而言，大学毕业后在传统的银行工作。后来，在数据爆发的时代，我有幸在国内数据最多的两家公司工作，我在阿里巴巴负责支付宝 BI 数据平台基础架构和应用架构，来到腾讯后一直负责腾讯的大数据业务。十多年的职业生涯，转换了公司，也转换了工作和生活的城市，但一直不变的是我的工作始终围绕着“数据”展开，无论是在传统 IT 行业，还是在互联网行业，“数据”始终是我工作的核心内容，而我自己最大的职业追求也离不开“数据”。

未来，在人们的生活中，数据将无时无刻无处不在，数据与商业的真正结合将爆发出强大的生命力和价值。作为服务于上层业务的基础支撑平台，最重要的地方在于技术的沉淀和积累，不断打磨优化。从技术研发人员的角度来讲，最重要的是修炼好自己的“内功”，不忘初心。最后，希望每一位读者都能够从本书中有所收获，练好数据的“内功”，与数据结缘。

蒋杰 博士
腾讯首席数据官、腾讯数据平台部总经理
CCF 大数据专家委员会委员
2018 年 7 月

推荐序 2

非常高兴收到了师弟朱锋、张博士和明哥关于 Spark SQL 的书稿，也非常荣幸被邀请为本书作友情序言。本人是朱锋在中科院软件所读博时的师兄，2014 年也曾经在明哥组内实习，目前在中科院从事大数据方面的科研工作，主要关注 Spark/Flink 等大数据处理系统及大数据分析算法，之前也在 GitHub 上写过 SparkInternals 等介绍技术原理的文章。

去年就听闻师弟他们在写一本关于 Spark SQL 的书，希望能够将生产环境中平台开发建设的一些经验总结出来。当时感觉有些惊讶，惊讶师弟从学术界到工业界能够快速转变，短时间内深入理解了整个系统，并能将经验总结成书。后来想了一下，这也是在情理之中的。朱锋的博士论文的内容就和 SQL-on-Hadoop 解决方案相关，在 Hadoop、Hive、HBase 等系统上也积累了多年的开发应用经验，在腾讯接触工业界实际案例后能够迅速应用所学知识，加上读博期间训练出来的抽象表达能力，与张博士一起，在明哥的全力支持下，足以写出一本有深度的技术图书。

虽然是熟人，但刚收到书时，我也有着顾虑。相信大多数读者也是如此，就是想知道这本书是否值得读，讲没讲干货，能否让读者快速理解 Spark SQL 的原理，能否对读者的实际工作有用。带着这些问题，我开始了阅读，发现越读干货越多，从 Spark SQL 历史到 Spark SQL 语法解析，从逻辑执行计划和物理执行计划生成，继续往下切入到 Tungsten 内存管理，全面讲述了 Spark SQL 技术的方方面面，不仅有原理介绍，还有实现细节的描述和总结，更有在海量数据和海量业务下的实践经验的总结。这些对想深入理解 Spark SQL，并对其进行优化改进，以及想更高效地使用 Spark SQL 的开发者、用户都有莫大的帮助。

抱着学习的态度读完本书，我在以下几个方面受益匪浅：逻辑执行计划、物理执行计划、优化方法等背后的技术原理，本书均透彻地讲清楚了。在读的过程中，我也在感慨，记得前几年带组里的师弟师妹们想在 Spark SQL 中添加关键词查询功能（类似 MySQL 中的 MATCH 关键词），发现需要在 Spark SQL 中添加新语法、改进语法解析、对翻译生成执行计划等一系列模块进行改进，由于当时缺少 Spark SQL 技术体系引导和深入解析的书籍，因此我们花了不少时间扎入代码，并自行总结技术体系和实现原理。本书的内容对于想对 Spark SQL 进行二次开发的读者非常有用，可以让二次开发事半功倍。

因为平时工作需要写论文，同时也会写技术报告和文档，所以深知写一本简洁易懂又包含复杂技术知识的图书有多困难，不仅需要花费大量精力阅读代码、分析细节，还需要在高层进行抽象总结、简洁表达。在阅读本书的过程中，可以从字里行间感受到作者们的用心和付出的努力。例如，作者在讲解 ANTLR 4 时，首先自己设计一个简单的例子，并进行相应的代码实现，在生成逻辑算子树时用到的访问者模式也会以实例进行说明。在技术展现方面，要画出图 3.8、图 7.10 等，不仅需要仔细阅读代码中的每个类的实现，而且需要考虑图形布局以达到直观的效果。

Spark 是一个易用性、通用性都很高的框架，除 Spark SQL 外，上层还有面向图计算的 GraphX 框架、机器学习 MLlib 库，流处理的 Spark Streaming 库，希望包括几位作者在内的业界专家能够为读者带来更多高水平的解析和总结。Spark SQL 本身也在演变中，希望本书有第二版、第三版等，能够不断加入更多的技术解析，不断完善。作为一名研究者，我自己也会努力去设计实现更多能够解决大数据实际问题的系统和方法，共勉之。

许利杰 (@JerryLead)
http://www.tcse.cn/~xulijie/
中科院软件所
2018 年 6 月 15 日

前言

极其迅速的信息传播将人们带入了大数据时代，也推动了大数据技术的发展。Spark 于 2009 年诞生于伯克利大学 AMP 实验室，至今已经形成完整的生态圈。除参与度高的开源社区外，各种相关的技术分享和论坛（如每年的 Spark Summit）也是如火如荼。得益于其灵活的 RDD 计算模型，Spark 系统高效地支持了各类应用，涉及 SQL 处理、图计算和机器学习等。

本书重点讲解 Spark SQL，该系统在企业中的应用非常广泛，也是 Spark 生态圈中较活跃的部分。从另一个视角来看，Spark SQL 是近年来 SQL-on-Hadoop 解决方案（包括 Hive、Presto 和 Impala 等）中的佼佼者，结合了数据库 SQL 处理和 Spark 分布式计算模型两个方面的技术，目标是取代传统的数据仓库。

在实际生产环境中，因为一些个性化的需求，往往涉及对原生的 Spark SQL 系统进行定制化的改造或新特性的添加，此过程需要开发人员对内部实现有深入的了解。然而笔者发现，目前业界在这方面的资料还比较缺乏，虽然已经涌现了一系列的文章和书籍，但内容通常都以 Spark 本身为主，或者停留在 API 使用和概括性介绍层面，难以满足开发人员的需求。

本书定位于弥补这方面的空白，对 Spark 的基本概念和功能（如 RDD 和调度等）不再展开讲解，而是将内容重点放在 SQL 内核实现的剖析上，旨在同读者一起“入于其中”，从源码实现上学习分布式计算和数据库领域的相关技术。

本书面向的读者

本书主要面向在企业中任职的系统架构师和软件开发人员，以及对大数据、分布式计算和数据库系统实现感兴趣的研究人员。需要注意的是，本书对读者在大数据系统和数据库方面的基础知识（SQL）上有一定的要求。对于初学者来说，最好能够首先参考相关资料，做到有所了解。

本书的主要内容

本书的内容可以分成 4 个部分：（1）背景和基础知识概述（第 1 ～ 2 章），这两章分别介绍 Spark SQL 的前生今世和与 Spark 相关的基础知识，对此熟悉的读者可以直接跳过；（2）Spark SQL 功能实现的各个阶段（第 3 ～ 6 章），这 4 章结合简单例子分别从整体和每个阶段的细节介绍内部机制，涉及 SQL 编译、逻辑计划和物理计划；（3）专题展开（第 7 ～ 10 章），这 4 章重点介绍 Spark SQL 中 Aggregation 和 Join 实现，深入分析 Tungsten 计划中的几项优化技术，以及 Spark SQL 连接 Hive 的实现；（4）实践部分（第 11 章），这一章分享 Spark SQL 系统在生产环境中的应用和一些改造优化经验。

一些约定和说明

- 相关术语：Spark 依赖于 JVM，主体采用 Scala 开发语言，部分功能也用到了 Java 语言来实现。本书没有严格地区分两种语言的术语，例如 Scala 语言中的 trait（特质）和 Java 语言中的 interface（接口）等，书中一般以 Java 语言的术语为主。此外，本书涉及的源码分析较多，不方便直接翻译的类或接口命名，以其英文命名为主。一些 SQL 关键词（例如 Select、Join 等）或 Spark 术语（例如 Shuffle、Partition、Executor 等）在不同上下文环境中也会出现大小写混用的情况。另外，Map 和 Reduce 虽然是来自 MapReduce 中的概念，但本书在介绍 Spark SQL 时也使用了这两个概念，分别用来表示 Shuffle 前的阶段和 Shuffle 后的阶段。
- 版本说明：本书使用的 Spark 版本是 Spark 2.X。笔者在写作时，以 2.1 版本和 2.2 版本中的实现机制为主。然而，Spark 社区活跃，版本的演化非常迅速（平均半年一个版本），读者在理解基本的框架和思路后可以结合 JIRA 上的相关 Issue 和对应的 Patch 进行跟踪。当然，后期最好的方式是参与到社区的贡献中。
- 推荐的阅读方式：本书内容涉及的实现细节较多，因此建议的阅读方式是结合代码进行理解。调试环境搭建好之后，在关键步骤插入日志信息，纵向（宏观）和横向（细节）分析交叉进行，最终做到在脑海中将上层的 SQL 语句映射为底层的 RDD 模型。

前沿技术的整理和分析并不是一件轻松的工作，从大纲的确定、内容的选择到最终出版得益于多方的大力支持。在此感谢电子工业出版社的各位编辑对本书出版提供的帮助，感谢马朋勃、马骉和邓飞等提出的宝贵修改建议。写作是一个不断学习并进行归纳和整理的过程，笔者在写作中也受到相关技术博客和论文思路的启发，在此一并感谢。

因笔者水平有限，本书的错漏和不足之处欢迎广大读者朋友批评指证。如果有更好的建议，也欢迎通过电子邮件联系几位作者：朱锋（wellfengzhu@gmail.com）、张韶全（shaoquan. zhang@hotmail.com）和黄明（andyyehoo@gmail.com）。

目　录

第1章 Spark SQL 背景

技术的诞生往往都有着特定的历史背景，而对技术来龙去脉的了解有助于我们从宏观层面把握全局方向。本章从大数据概念产生以来 10 多年的技术发展轨迹讲起，简要回顾 Spark SQL 的演化历程。

1.1 大数据与 Spark 系统

大数据一词，最早出现于 20 世纪 90 年代，由数据仓库之父 Bill Inmon 所提及。2008 年，*Nature* 杂志出版了大数据专刊“Big Data”，专门讨论海量数据对互联网、经济、环境和生物等各方面的影响与挑战。2011 年，*Science* 出版了如何应对数据洪流（Data deluge）的专刊“Dealing with Data”，指出如何利用大数据中宝贵的数据价值来推动人类社会的发展。迄今为止，大数据并没有统一的标准定义，业界和学术界通常用 5 个方面的属性（5V）来描述大数据的特点：Volume（体量大）、Velocity（时效高）、Variety（类型多）、Veracity（真实性）、Value（价值大）。

大数据一方面意味着巨大的信息价值，另一方面也带来了技术上的挑战，使得传统的计算机技术难以在合理的时间内达到数据存储、处理和分析的目的。大数据应用的爆发性增长，已经衍生出独特的架构，并直接推动了存储、网络和计算技术的研究。Google 公司于 2003 年在 SOSP 会议上发表论文介绍分布式文件系统 GFS[1]，于 2004 年在 OSDI 会议上发表论文介绍分布式大数据编程模型与处理框架 MapReduce[2]，于 2006 年再次在 OSDI 会议上发表论文介绍分布式数据库 BigTable[3] 的实现。以上三者统称为 Google 公司初期大数据技术的“三驾马车”，自此各种大数据存储与处理技术开始蓬勃发展。

Spark[4] 分布式计算框架是大数据处理领域的佼佼者，由美国加州大学伯克利分校的 AMP 实验室开发。相比于流行的 Hadoop[5] 系统，Spark 优势明显。Spark 一方面提供了更加灵活丰富的数据操作方式，有些需要分解成几轮 MapReduce 作业的操作，可以在 Spark 里一轮实现；另一方面，每轮的计算结果都可以分布式地存放在内存中，下一轮作业直接从内存中读取上一轮的数据，节省了大量的磁盘 IO 开销。因此，对于机器学习、模式识别等迭代型计算，Spark

在计算速度上通常可以获得几倍到几十倍的提升。得益于 Spark 对 Hadoop 计算的兼容，以及对迭代型计算的优异表现，成熟之后的 Spark 系统得到了广泛的应用。例如，在大部分公司中，典型的场景是将 Hadoop（HDFS）作为大数据存储的标准，而将 Spark 作为计算引擎的核心。

经过多年的发展，Spark 已成为目前大数据处理领域炙手可热的顶级开源项目。屈指一算，Spark 从诞生到 2018 年，已经走过了整整 9 个年头。如图 1.1 所示，第一个版本的 Spark 诞生在 2009 年，代码量仅有 3900 行左右，其中还包含 600 行的例子和 300 多行的测试代码。当时，Hadoop 在国外已经开始流行，但是其 MapReduce 编程模型较为笨拙和烦琐，Matei 借鉴 Scala 的 Collection 灵感，希望开发出一套能像操作本地集合一样简捷、高效操作远程大数据的框架，并能运行于 Mesos[6] 平台上，于是就有了 Spark 最初的 0.1 版本和 0.2 版本。后来，Reynold Xin 加入这个项目，在协助对 Core 模块进行开发的同时，在其之上启动了 Shark[7] 项目，希望能够让 Spark 更好地处理 SQL 任务，替代当时流行的 Hive[8]（基于 Hadoop 的数据仓库解决方案）。当然，这个版本的 Shark 在多个方面都存在先天的不足，Spark 在后来的发展过程中将其废弃，另起炉灶，从头来过，这也就是众所周知的 Spark SQL[9]，相关细节会在后面进一步详述。

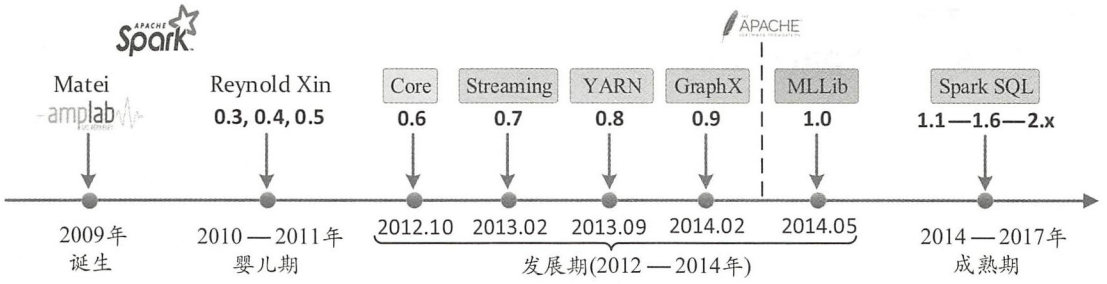


图 1.1　Spark 发展历程

在此期间，Spark 经历了一个蓬勃的生长期，从 2012 年的 0.6 版本开始，Core 模块开始趋于稳定，接近生产级别，不再是实验室的产物。我国的阿里巴巴团队开始将其用于线上正式作业，并取得了比使用 MapReduce 更好的效果。同时，Intel 公司也开始投入力量到该项目的开发中。在 0.7 版本中，Tathagata Das 开始加入 Streaming 模块，使得 Spark 具备准实时的流处理能力。到了 0.8 版本，Spark 正式支持在 YARN[10] 上的部署运行，Yahoo 公司贡献了重要的代码，国内的阿里巴巴团队也开始将其正式部署到内部的云梯集群，搭建了 300 台专用 Spark 集群。在 0.9 版本中，图处理系统 GraphX[11] 正式成为独立模块，同年，Apache 接受 Spark 成为顶级项目，从孵化期到正式项目，只经历了短短半年时间，这种速度在 Apache 开源社区是非常难得的。到了 1.0 版本，孟祥瑞等人加入 DataBricks 公司，主导的 MLLib[12] 也成为正式模块。至此，各个主要模块形成了较完整的 Spark 技术栈（生态系统），如图 1.2 所示。

可以看出，在 2012—2014 年，Spark 经历了一个高速的发展过程，各个模块快速演进，各大公司和全球顶尖开发人员的加入，使得整个项目充满生命力和活力。DataBricks 公司成立后，

多数客户的需求集中在常用的数据处理方面，而这需要 Spark 系统有完善且强大的 SQL 能力。因此，在 2014—2017 年，Spark 技术栈重点关注 Spark SQL 子项目，在钨丝计划（Tungsten）的基础上，开始了 DataFrame 和 DataSet 为用户接口核心的 SQL 功能开发，使得 Spark SQL 项目发展迅速，整体内核也针对 SQL 做了很多优化。从 1.6 版本开始，社区的发展一步一个脚印，到如今功能完善的 2.x 版本，Spark SQL 已经非常成熟，完全达到了商用的程度。

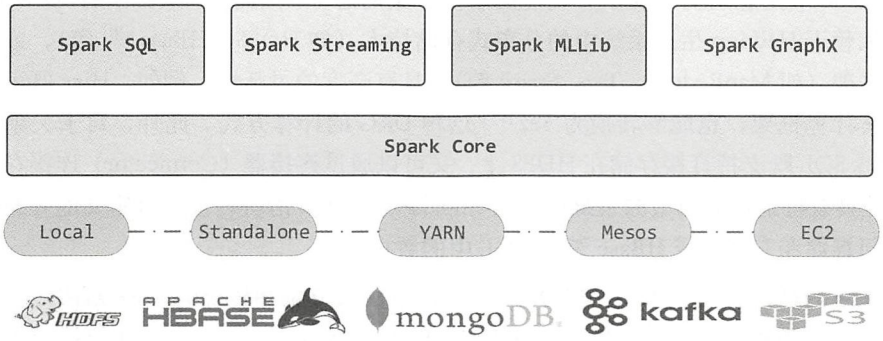


图 1.2 Spark 技术栈

1.2 关系模型与 SQL 语言

计算机软件系统离不开底层的数据，而数据的存储管理需求促进了数据库技术的发展。早期的数据库包含网状数据库和层次数据库等不同类型。在 20 世纪 70 年代，IBM 的研究员 E.F.Codd 提出关系模型[13]。因为具有严格的数学理论基础、较高的抽象级别，而且便于理解和使用，所以关系模型成为现代数据库产品的主流，并占据统治地位 40 多年。此外，IBM 公司研究人员将关系模型中的数学准则以关键字语法表现出来，于 1974 年里程碑式地提出了 SQL（Structured Query Language）语言[14]。SQL 语言是一种声明式的语言，提供了查询、操纵、定义和控制等数据库生命周期中的全部操作。另外，建立在关系模型之上并提供 SQL 语言支持的关系数据库管理系统（如 Oracle、DB2、SQL Server 等），已经成为企业通用的数据存储解决方案。对数据管理人员和应用开发人员来说，关系模型和 SQL 语言一直是必不可少的技术基础。

尽管非结构化数据和半结构化数据在数据量上占有绝大部分比例，但结构化数据和 SQL 需求仍旧具有举足轻重的作用。当 Hadoop 生态系统进入企业时（注：广义上 Spark 也可以看作是 Hadoop 生态系统中的一员），必须面对的一个问题就是怎样解决和应对传统成熟的信息架构。在企业内部，如何处理原有的结构化数据是企业进入大数据领域所面临的难题。Hadoop 生态系统的初衷在于解决日志文件分析、互联网点击流、倒排索引和大文件存储等非结构化数据的问题。在此背景的驱动下，面向 Hadoop 生态系统的 SQL 查询处理技术及框架（统称为

“SQL-on-Hadoop”[15-17]）应运而生。SQL-on-Hadoop 解决方案为用户提供关系模型和 SQL 查询接口，并透明地将存储与查询转换为 Hadoop 生态系统的对应技术来管理海量结构化数据。

作为数据分析领域重要的支撑，SQL-on-Hadoop 成为近几年来广受关注的热点，并涌现出大量的产品，如典型的 Hive、Impala[18] 和 Presto[19] 等。所以，从横向来看，Spark SQL 算是 SQL-on-Hadoop 解决方案大家庭中的重要一员。SQL-on-Hadoop 并非专指某一个特定的系统，而是借助于 Hadoop 生态系统完成 SQL 功能的解决方案的总称。因此，一个具体的 SQL-on-Hadoop 系统往往依赖于 Hadoop 生态系统中的分布式存储技术（如 HDFS、HBase[20] 等），或者利用分布式计算框架（如 MapReduce、Tez、Spark 等），具有高度的灵活性。例如，Hive 既可以转换为 MapReduce 计算框架，也能够转换为 Tez[21] 这种 DAG 的计算方式。此外，对于关系数据表的访问，Spark SQL 既支持直接存储在 HDFS 上，又可以通过连接器（Connector）连接存储在类似 HBase 这种具有特定数据模型的系统中。而 Impala 则将 SQL 语言转换为自定义的分布式计算框架，来访问存储在 HDFS 或 HBase 等 NoSQL 中的数据。

需要注意的是，Spark SQL 这类 SQL-on-Hadoop 解决方案和传统的 MPP 解决方案在架构上存在很大差异。根据 Hadoop 生态系统的特点，SQL-on-Hadoop 解决方案从架构上来看可以简单地划分为三层结构。最上层是应用（语言）层，应用层为用户提供数据管理查询的接口，不同的 SQL-on-Hadoop 系统往往提供各自的 SQL 语法特性，如 Hive 的 HiveQL、Pig 的 PigLatin[22] 和 Spark SQL 的 DataFrame 等。在大数据场景下，应用层也包含一些针对特别需求的接口，如 BlinkDB[23] 所支持的近似查询功能等。应用层之下是分布式执行层，SQL-on-Hadoop 系统通过一定的规则或策略将 SQL 语句转换为对应的计算模型。除 MapReduce、Spark 等通用的计算框架外，分布式执行层可能是某些系统自定义的计算单元，例如 Impala 中的 Query Exec Engine 等。分布式执行层通过接口访问数据存储层中的数据，并完成相应的计算任务。SQL-on-Hadoop 系统的底层是数据存储层，主要负责对关系数据表这样的逻辑视图进行存储与管理。目前，各种 SQL-on-Hadoop 数据存储层基本都支持分布式文件系统 HDFS 和分布式 NoSQL 数据库。总的来看，SQL-on-Hadoop 解决方案类似“堆积木”，各层之间松耦合，并可以灵活组合。数据存储层与分布式执行层之间通过特定的数据读写接口进行数据的交互。这种分层解耦的方式一方面具有通用性（各层可以分别分离为子系统）和灵活性（彼此能够互相组合）的优势，另一方面隔离了各层的特性，限制了深度集成优化的空间。

1.3 Spark SQL 发展历程

Spark SQL 的前身是 Shark[7]，即“Hive on Spark”，由 Reynold Xin 主导开发。Shark 项目最初启动于 2011 年，当时 Hive 几乎算是唯一的 SQL-on-Hadoop 选择方案。Hive 将 SQL 语句翻译为 MapReduce，正如前文所提到的，性能会受限于 MapReduce 计算模型，始终无法满足各种交

互式 SQL 分析的需求，因此许多机构仍然依赖传统的企业数据仓库（EDW）。Shark 的提出就是针对这种需求的，目标是既能够达到 EDW 的性能，又能够具有 MapReduce 的水平扩展功能。

Shark 建立在 Hive 代码的基础上，只修改了内存管理、物理计划、执行 3 个模块中的部分逻辑。Shark 通过将 Hive 的部分物理执行计划交换出来（“swapping out the physical execution engine part of Hive”），最终将 HiveQL 转换为 Spark 的计算模型，使之能运行在 Spark 引擎上，从而使得 SQL 查询的速度得到 10 ～ 100 倍的提升。此外，Shark 的最大特性是与 Hive 完全兼容，并且支持用户编写机器学习或数据处理函数，对 HiveQL 执行结果进行进一步分析。

但是，随着 Spark 的不断发展，Shark 对 Hive 的重度依赖体现在架构上的瓶颈越来越突出。一方面，Hive 的语法解析和查询优化等模块本身针对的是 MapReduce，限制了在 Spark 系统上的深度优化和维护；另一方面，过度依赖 Hive 制约了 Spark 的“One Stack Rule Them All”既定方针，也制约了技术栈中各个组件的灵活集成。在此背景下，Spark SQL 项目被提出来，由 Michael Armbrust 主导开发。Spark SQL 抛弃原有 Shark 的架构方式，但汲取了 Shark 的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive 兼容性等，重新开发了 SQL 各个模块的代码。由于摆脱了对 Hive 的依赖，Spark SQL 在数据兼容、性能优化、组件扩展方面都得到了极大的提升。在 2014 年 7 月 1 日的 Spark 峰会上，Databricks 公司宣布终止对 Shark 的开发，将后续重点放到 Spark SQL 上。Spark SQL 涵盖了 Shark 的所有特性，用户可以从 Shark 进行无缝升级，至此，Shark 的发展画上了句号。Spark SQL 开始迎来蓬勃的发展阶段。如今，Spark SQL 已经成为 Apache Spark 中最为活跃的子项目。

1.4 本章小结

尽管各式各样的计算模型、处理框架、平台系统层出不穷，令人眼花缭乱难以选择，但是万变不离其宗，底层所涉及的分布式技术、编译原理、数据库理论等基础知识都是通用的。作为开发人员，抓住这些系统背后通用的核心技术才是根本。

本书写作的目的即是如此，旨在从技术细节上深入剖析一个典型系统的实现，并在此基础上分享一些开发实践的经验。Spark SQL 作为 Spark 技术体系中最为活跃的子项目，发展迅速，其内部实现也越来越复杂，许多技术具有广泛的借鉴意义。本书面向大数据开发人员和期望对内部原理有深入了解的用户，内容以 Spark 2.1/2.2 版本为主，注重源码层面细节的分析，希望本书能为读者的大数据之旅提供帮助。

第2章 Spark基础知识介绍

作为学习Spark SQL的预备内容，本章用少量篇幅讲解Spark系统的基础知识，包括RDD编程模型、DataFrame和Dataset用户接口。目前关于Spark技术的文章和书籍有很多，熟悉的读者可以直接跳过本章。

2.1 RDD编程模型

编程模型的灵活程度决定了分布式系统的适用范围。实际上在Spark被提出之前，业界已经出现了大量的分布式计算框架，并提供了各种编程接口抽象（Abstraction）。例如，在典型的MapReduce编程模型中，用户只需要编写Map和Reduce接口即可实现业务逻辑；又如Apache的Hama[24]系统，直接提供了BSP模型的编程接口。

虽然在编程接口的种类和丰富程度上已经比较完善了，但这些系统普遍都缺乏操作分布式内存的接口抽象，导致很多应用在性能上非常低效。这些应用的共同特点是需要在多个并行操作之间重用工作数据集，典型的场景就是机器学习和图应用中常用的迭代算法（每一步对数据执行相似的函数）。例如，许多机器学习算法需要将当前迭代权值调优后的结果数据集作为下次迭代的输入，在使用MapReduce计算框架经过一次Reduce操作后输出数据会写回磁盘，然后从磁盘读取作为下次迭代的输入，这种密集的磁盘IO操作方式极大地降低了性能。

针对这类数据重用的需求，研发人员也尝试过各种解决方案。例如，Google的迭代式图计算系统Pregel[25]会将中间数据存储在内存中，HaLoop[26]在基本的MapReduce接口上又扩展了一个专门用于迭代的接口。然而，这些方案都存在很大的局限性，针对的也只是特定的计算需求，数据重用隐藏在系统实现背后，没有将重用逻辑显式地抽象出来形成通用接口。

RDD[27]则是直接在编程接口层面提供了一种高度受限的共享内存模型，如图2.1所示。RDD是Spark的核心数据结构，全称是弹性分布式数据集（Resilient Distributed Dataset），其本质是一种分布式的内存抽象，表示一个只读的数据分区（Partition）集合。一个RDD通常只能通过其他的RDD转换而创建。RDD定义了各种丰富的转换操作（如map、join和filter等），通

过这些转换操作，新的 RDD 包含了如何从其他 RDD 衍生所必需的信息，这些信息构成了 RDD 之间的依赖关系（Dependency）。依赖具体分为两种，一种是窄依赖，RDD 之间分区是一一对应的；另一种是宽依赖，下游 RDD 的每个分区与上游 RDD（也称之为父 RDD）的每个分区都有关，是多对多的关系。窄依赖中的所有转换操作可以通过类似管道（Pipeline）的方式全部执行，宽依赖意味着数据需要在不同节点之间 Shuffle 传输。

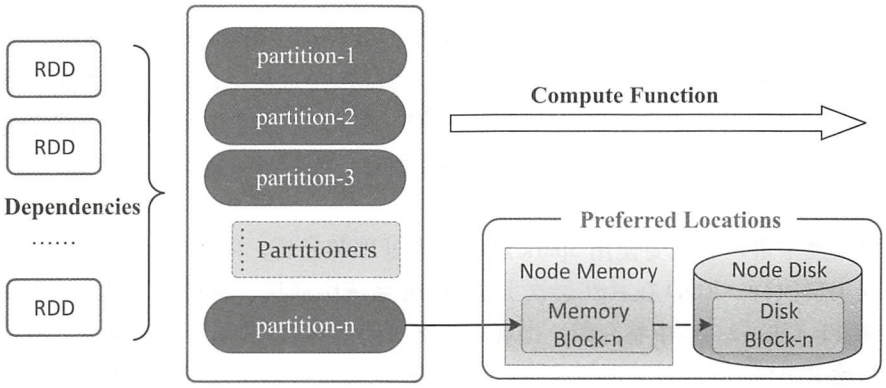


图 2.1 RDD 核心抽象

RDD 计算的时候会通过一个 compute 函数得到每个分区的数据。若 RDD 是通过已有的文件系统构建的，则 compute 函数读取指定文件系统中的数据；如果 RDD 是通过其他 RDD 转换而来的，则 compute 函数执行转换逻辑，将其他 RDD 的数据进行转换。RDD 的操作算子包括两类，一类是 transformation，用来将 RDD 进行转换，构建 RDD 的依赖关系；另一类称为 action，用来触发 RDD 的计算，得到 RDD 的相关计算结果或将 RDD 保存到文件系统中。

在 Spark 中，RDD 可以创建为对象，通过对象上的各种方法调用来对 RDD 进行转换。经过一系列的 transformation 逻辑之后，就可以调用 action 来触发 RDD 的最终计算。通常来讲，action 包括多种方式，可以是向应用程序返回结果（show、count 和 collect 等），也可以是向存储系统保存数据（saveAsTextFile 等）。在 Spark 中，只有遇到 action，才会真正地执行 RDD 的计算（注：这被称为惰性计算，英文为 Lazy Evqluation），这样在运行时可以通过管道的方式传输多个转换。

总结而言，基于 RDD 的计算任务可描述为：从稳定的物理存储（如分布式文件系统 HDFS）中加载记录，记录被传入由一组确定性操作构成的 DAG（有向无环图），然后写回稳定存储。RDD 还可以将数据集缓存到内存中，使得在多个操作之间可以很方便地重用数据集。总的来讲，RDD 能够很方便地支持 MapReduce 应用、关系型数据处理、流式数据处理（Stream Processing）和迭代型应用（图计算、机器学习等）。

在容错性方面，基于 RDD 之间的依赖，一个任务流可以描述为 DAG。在实际执行的时候，RDD 通过 Lineage 信息（血缘关系）来完成容错，即使出现数据分区丢失，也可以通过 Lineage 信息重建分区。如果在应用程序中多次使用同一个 RDD，则可以将这个 RDD 缓存起来，该 RDD 只有在第一次计算的时候会根据 Lineage 信息得到分区的数据，在后续其他地方用到这个 RDD 的时候，会直接从缓存处读取而不用再根据 Lineage 信息计算，通过重用达到提升性能的目的。虽然 RDD 的 Lineage 信息可以天然地实现容错（当 RDD 的某个分区数据计算失败或丢失时，可以通过 Lineage 信息重建），但是对于长时间迭代型应用来说，随着迭代的进行，RDD 与 RDD 之间的 Lineage 信息会越来越长，一旦在后续迭代过程中出错，就需要通过非常长的 Lineage 信息去重建，对性能产生很大的影响。为此，RDD 支持用 checkpoint 机制将数据保存到持久化的存储中，这样就可以切断之前的 Lineage 信息，因为 checkpoint 后的 RDD 不再需要知道它的父 RDD，可以从 checkpoint 处获取数据。

从用户角度来讲，如果要使用 Spark 进行数据处理和分析，需要编写一个 Driver 程序，并将其提交到集群执行。RDD 在 API 层次上是 Spark 系统中底层，且最为灵活，自从 DataFrame 和 Dataset 接口（注：2.2 节中将介绍）出现后，直接使用 RDD 的场景越来越少。但是，作为底层支持，熟悉 RDD 的原理对于理解 Spark SQL 物理执行阶段的实现逻辑非常重要。如图 2.2 所示是常见的 WordCount 案例。

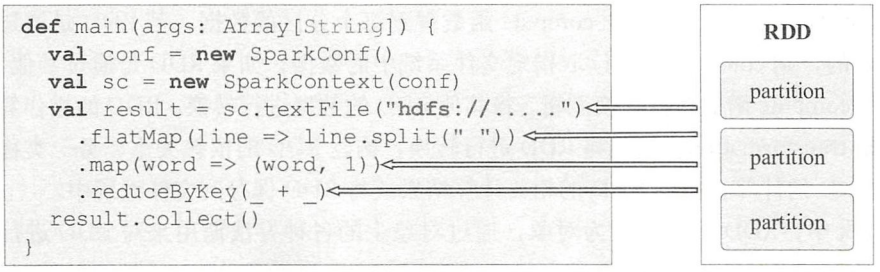


图 2.2　RDD 编程模型（WordCount 案例）

众所周知，这个简单的应用程序用来统计一个数据集中每个单词出现的次数。在 Spark 的程序实现中，首先构造 SparkContext 对象；然后调用 textFile 方法从 HDFS 中加载数据以得到初始的 RDD（记为 rdd_1），其中每条记录为数据中的一行句子；接下来调用 flatMap 方法进行 RDD 的转换，将一行句子切分为多个独立的单词，得到新的 RDD（记为 rdd_2）；再通过 map 方法将每个词映射为 key-value 形式，其中 key 为词本身，value 为初始计数值 1，再次得到一个新的 RDD（记为 rdd_3）；最后，采用 reduceByKey 方法将 rdd_3 中的所有记录归并，统计得到每个单词的计数，转换为最终的 RDD（记为 rdd_4）。这里需要注意，rdd_1 到 rdd_4 都不会触发执行，直到调用 collect 这个操作后，整个 DAG 才开始调度执行。

2.2 DataFrame 与 Dataset

对于数据分析开发人员来说，一套直观易用且富有表达力的 API 能够极大地提高生产力。Spark 在 RDD 基础上，提供了 DataFrame 和 Dataset 用户编程接口，并且在跨语言（Scala、Java、Python 和 R）方面具有很好的支持。为了追求简化，降低开发人员的学习成本，从 Spark 2.0 开始，DataFrame 和 Dataset 进行了统一。

DataFrame 与 RDD 一样，都是不可变分布式弹性数据集。不同之处在于，RDD 中的数据不包含任何结构信息，数据的内部结构可以被看作“黑盒”，因此，直接使用 RDD 时需要开发人员实现特定的函数来完成数据结构的解析；而 DataFrame 中的数据集类似于关系数据库中的表，按列名存储，具有 Schema 信息，开发人员可以直接将结构化数据集导入 DataFrame。DataFrame 的数据抽象是命名元组（对应 Row 类型），相比 RDD 多了数据特性，因此可以进行更多的优化。例如，使用 DataFrame 来实现 WordCount，构造 RDD 之后，直接用 toDF 得到只有一列（列名为“line”）的 DataFrame，然后调用 explode 方法将一行数据展开，最后使用 groupBy 方法完成聚合逻辑。

```
val linesDF = sc.textFile("hdfs://...").toDF("line")
val wordsDF = linesDF.explode("line", "word")((line: String) => line.split(" "))
val wordCountDF = wordsDF.groupBy("word").count()
wordCountDF.collect()
```

Dataset 是比 DataFrame 更为强大的 API，如图 2.3 所示，两者整合之后 DataFrame 本质上是一种特殊的 Dataset（Dataset[Row] 类型）。Dataset 具有两个完全不同的 API 特征：强类型（Strongly-Typed）API 和弱类型（Untyped）API。强类型一般通过 Scala 中定义的 Case Class 或 Java 中的 Class 来指定。

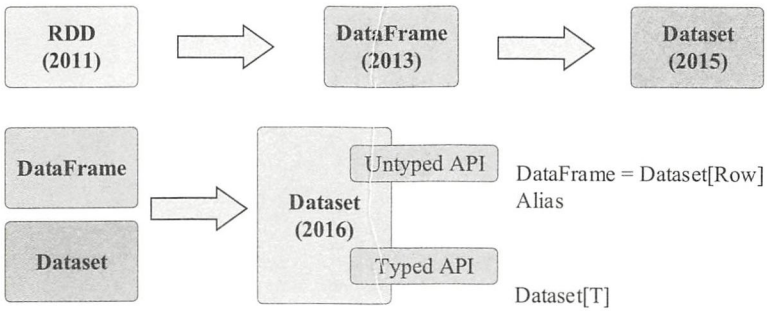


图 2.3 DataFrame 与 Dataset

作为 DataFrame 的扩展，Dataset 结合了 RDD 和 DataFrame 的优点，提供类型安全和面向对象的编程接口，并引入了编码器（Encoder）的概念。典型的 Dataset 创建与使用案例如下，其中定义的 Person 类就起到了 Encoder 的作用。在映射的过程中，Encoder 首先检查定义的 Person 类的类型是否与数据相符，如果不相符（例如 age 字段大于 Long 的最大值等），则能够及时提供有用的错误信息，防止以不正确的方式处理数据。类似于 DataFrame，Dataset 创建之后也能够很容易地使用 lambda 表达式进行各种转换操作。

```
case class Person(name: String, age: Long)
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
```

Encoder 不仅能够在编译阶段完成类型安全检查，还能够生成字节码与堆外数据进行交互，提供对各个属性的按需访问，而不必对整个对象进行反序列化操作，极大地减少了网络数据传输的代价。此外，DataFrame 和 Dataset 这些高级的 API，能够利用 Spark SQL 中的 Optimizer 和 Tungsten 技术自动完成存储和计算的优化，降低内存使用，并极大地提升性能。

2.3 本章小结

本章简单地介绍了 Spark 系统架构和编程接口中的基本概念。DataFrame 从 API 上借鉴了 R 和 Python 语言中 Pandas 的 DataFrame 的概念，是业界标准结构化数据处理接口。严格来讲，DataFrame 和 Dataset 都属于 Spark SQL 模块，每个操作都可以被看作是一条完整 SQL 语句的“零碎”逻辑。对这些基本概念的了解有助于理解后续内容。从下一章开始，将分析 Spark SQL 底层的实现与优化机制。

第3章 Spark SQL 执行全过程概述

从高层的语言到底层的计算模型通常都包含各种复杂的中间转换。具体到 SQL 语言，在数据库几十年的发展过程中，整个架构体系已经非常成熟，各种相似系统（如 Hive 和 Impala 等）的实现均大同小异。本章通过一个简单的案例对整个执行过程进行概述，并简单介绍 Spark SQL 内部机制中涉及的基本概念和数据结构。

3.1 从 SQL 到 RDD：一个简单的案例

在典型的 Spark SQL 应用场景中，数据的读取、数据表的创建和分析都是必不可少的过程。通常来讲，SQL 查询所面对的数据模型以关系表为主。如图 3.1 所示的案例显示了使用 Spark SQL 进行数据分析的一般步骤。

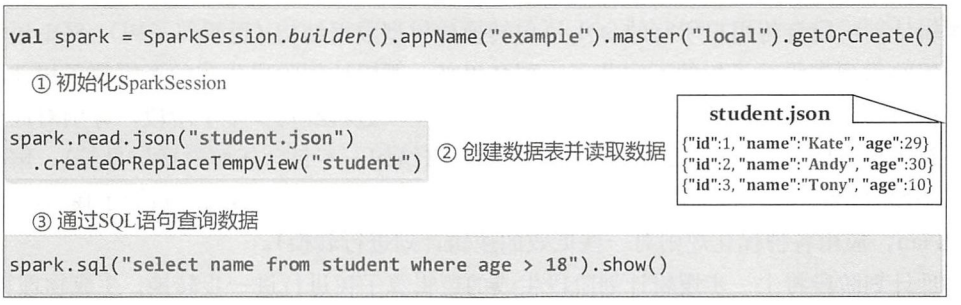


图 3.1　一个简单的案例

案例中涉及的操作分为 3 步。

（1）创建 SparkSession 类。从 2.0 版本开始，SparkSession 逐步取代 SparkContext 成为 Spark 应用程序的入口。

（2）创建数据表并读取数据。这里假设数据存储在本地名为 student 的 json 文件中，包含

3 条记录且每条记录包含 3 个列（分别对应学生的 id、name 和 age）。本案例创建了同样名为 student 的数据表（视图）。

（3）通过 SQL 进行数据分析。在 SparkSession 类的 sql 方法中可以输入任意满足语法的语句，本案例所查询的数据是年龄在 18 岁以上的学生名字。

值得一提的是，上述案例第 2 步创建数据表时虽然没有显示调用 SQL 语句（如关系数据库中的“create table”等），但其本质上也是 SQL 中的一种（DDL 操作），在内部转换执行时，所涉及的流程和第 3 步执行 SQL 查询的流程类似。因此，从一般性考虑，后续内容只对第 3 步背后的实现进行分析。

这里首先从通用的角度介绍 SQL 转换的过程。一般来讲，对于 Spark SQL 系统，从 SQL 到 Spark 中 RDD 的执行需要经过两个大的阶段，分别是逻辑计划（LogicalPlan）和物理计划（PhysicalPlan），如图 3.2 所示。

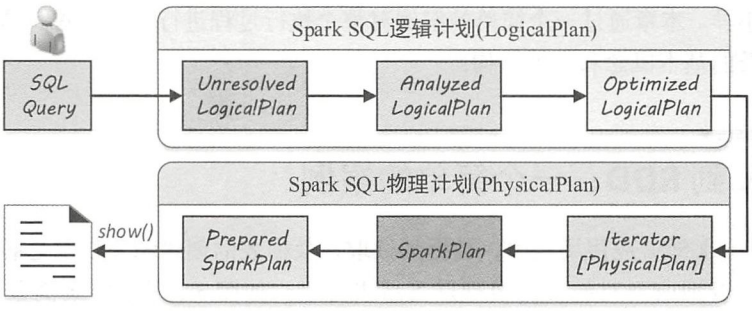


图 3.2　SQL 执行全过程概览

逻辑计划阶段会将用户所写的 SQL 语句转换成树型数据结构（逻辑算子树），SQL 语句中蕴含的逻辑映射到逻辑算子树的不同节点。顾名思义，逻辑计划阶段生成的逻辑算子树并不会直接提交执行，仅作为中间阶段。最终逻辑算子树的生成过程经历 3 个子阶段，分别对应未解析的逻辑算子树（Unresolved LogicalPlan，仅仅是数据结构，不包含任何数据信息等）、解析后的逻辑算子树（Analyzed LogicalPlan，节点中绑定各种信息）和优化后的逻辑算子树（Optimized LogicalPlan，应用各种优化规则对一些低效的逻辑计划进行转换）。

物理计划阶段将上一步逻辑计划阶段生成的逻辑算子树进行进一步转换，生成物理算子树。物理算子树的节点会直接生成 RDD 或对 RDD 进行 transformation 操作（注：每个物理计划节点中都实现了对 RDD 进行转换的 execute 方法）。同样地，物理计划阶段也包含 3 个子阶段：首先，根据逻辑算子树，生成物理算子树的列表 Iterator[PhysicalPlan]（同样的逻辑算子树可能对应多个物理算子树）；然后，从列表中按照一定的策略选取最优的物理算子树（SparkPlan）；最后，对选取的物理算子树进行提交前的准备工作，例如，确保分区操作正确、物理算子树节点重用、执行代码生成等，得到“准备后”的物理算子树（Prepared SparkPlan）。经过上述步骤后，

物理算子树生成的 RDD 执行 action 操作（如例子中的 show），即可提交执行。

从 SQL 语句的解析一直到提交之前，上述整个转换过程都在 Spark 集群的 Driver 端进行，不涉及分布式环境。SparkSession 类的 sql 方法调用 SessionState 中的各种对象，包括上述不同阶段对应的 SparkSqlParser 类、Analyzer 类、Optimizer 类和 SparkPlanner 类等，最后封装成一个 QueryExecution 对象。因此，在进行 Spark SQL 开发时，可以很方便地将每一步生成的计划单独剥离出来分析。

回到前面的案例，SQL 语句较为简单（不包含 Join 和 Aggregation 等操作），因此其转换过程也相对简单。如图 3.3 所示，左上角是 SQL 语句，生成的逻辑算子树中有 Relation、Filter 和 Project 节点，分别对应数据表、过滤逻辑（age>18）和列剪裁逻辑（只涉及 3 列中的 2 列）。下一步的物理算子树从逻辑算子树一对一映射得到，Relation 逻辑节点转换为 FileSourceScanExec 执行节点，Filter 逻辑节点转换为 FilterExec 执行节点，Project 逻辑节点转换为 ProjectExec 执行节点。

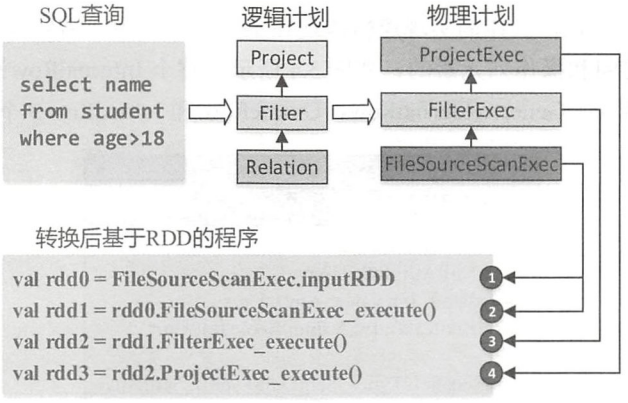


图 3.3 实际转换过程

生成的物理算子树根节点是 ProjectExec，每个物理节点中的 execute 函数都是执行调用接口，由根节点开始递归调用，从叶子节点开始执行。图 3.3下方展示了物理算子树的执行逻辑，与直接采用 RDD 进行编程类似。需要注意的是，FileSourceScanExec 叶子执行节点中需要构造数据源对应的 RDD，FilterExec 和 ProjectExec 中的 execute 函数对 RDD 执行相应的 transformation 操作。

总的来看，SQL 转换为 RDD 在流程上比较清晰。虽然实际生产环境中的 SQL 语句非常复杂，涉及的映射操作也比较烦琐，但总体上仍然遵循上述步骤。在后续章节会详细剖析这一整套转换流程。第 4 章会介绍如何从 SQL 解析为逻辑计划，第 5 章对逻辑计划（LogicalPlan）中的各个子阶段进行详细分析，第 6 章介绍物理计划（PhysicalPlan）中的实现机制。

3.2 重要概念

Spark SQL 内部实现上述流程中平台无关部分的基础框架称为 Catalyst。在深入分析流程每个阶段的原理之前，本节先简要介绍 Catalyst 中涉及的重要概念和数据结构，主要包括 InternalRow 体系、TreeNode 体系和 Expression 体系。

3.2.1 InternalRow 体系

数据处理首先需要考虑如何表示数据。对于关系表来讲，通常操作的数据都是以“行”为单位的。在 Spark SQL 内部实现中，InternalRow 就是用来表示一行行数据的类，因此图 3.3 中物理算子树节点产生和转换的 RDD 类型即为 RDD[InternalRow]。此外，InternalRow 中的每一列都是 Catalyst 内部定义的数据类型。

从类的定义来看，InternalRow 作为一个抽象类，包含 numFields 和 update 方法，以及各列数据对应的 get 与 set 方法，但具体的实现逻辑体现在不同的子类中。需要注意的是，InternalRow 中都是根据下标来访问和操作列元素的。如图 3.4 所示，整个 InternalRow 体系比较简单，其具体的实现不多，包括 BaseGenericInternalRow、UnsafeRow 和 JoinedRow 3 个直接子类。

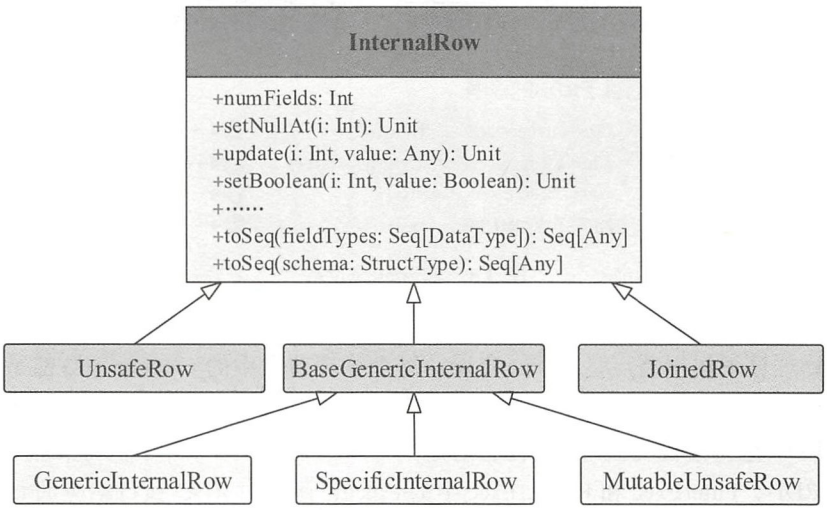


图 3.4　InternalRow 体系

- BaseGenericInternalRow：同样是一个抽象类，实现了 InternalRow 中定义的所有 get 类型方法，这些方法的实现都通过调用类中定义的 genericGet 虚函数进行，该函数的实现在下一级子类中。

- JoinedRow：顾名思义，该类主要用于 Join 操作，将两个 InternalRow 放在一起形成新的 InternalRow。使用时需要注意构造参数的顺序。
- UnsafeRow：不采用 Java 对象存储的方式，避免了 JVM 中垃圾回收（GC）的代价。此外，UnsafeRow 对行数据进行了特定的编码，使得存储更加高效。作为 Tungsten 计划的重要内容，相关实现在第 9 章中会涉及。

从直接子类继续往下，BaseGenericInternalRow 也衍生出 3 个子类，分别是 GenericInternalRow、SpecificInternalRow 和 MutableUnsafeRow 类。其中，MutableUnsafeRow 和 UnsafeRow 相关，用来支持对特定的列数据进行修改，这里暂时不作介绍。下面主要介绍 GenericInternalRow 和 SpecificInternalRow。

```
class GenericInternalRow(val values: Array[Any]) extends BaseGenericInternalRow {
  protected def this() = this(null)
  def this(size: Int) = this(new Array[Any](size))
  override protected def genericGet(ordinal: Int) = values(ordinal)
  override def toSeq(fieldTypes: Seq[DataType]): Seq[Any] = values.clone()
  override def numFields: Int = values.length
  override def setNullAt(i: Int): Unit = { values(i) = null}
  override def update(i: Int, value: Any): Unit = { values(i) = value }
  override def copy(): GenericInternalRow = this
}
```

阅读以上代码，可见 GenericInternalRow 构造参数是 Array[Any] 类型，采用对象数组进行底层存储，genericGet 也是直接根据下标访问的。这里需要注意，数组是非拷贝的，因此一旦创建，就不允许通过 set 操作进行改变。而 SpecificInternalRow 则是以 Array[MutableValue] 为构造参数的，允许通过 set 操作进行修改。

3.2.2 TreeNode 体系

从案例的转换过程可以看到，无论是逻辑计划还是物理计划，都离不开中间数据结构。在 Catalyst 中，对应的是 TreeNode 体系。TreeNode 类是 Spark SQL 中所有树结构的基类，定义了一系列通用的集合操作和树遍历操作接口。

如图 3.5 所示，TreeNode 内部包含一个 Seq[BaseType] 类型的变量 children 来表示孩子节点。TreeNode 定义了 foreach、map、collect 等针对节点操作的方法，以及 transformUp 和 transformDown 等遍历节点并对匹配节点进行相应转换的方法。TreeNode 本身是 scala.Product 类型，因此可以通过 productElement 函数或 productIterator 迭代器对 Case Class 参数信息进行索引和遍历。实际上，TreeNode 一直在内存里维护，不会 dump 到磁盘以文件形式存储，且无论在映射逻辑执行计划阶段，还是优化逻辑执行计划阶段，树的修改都是以替换已有节点的方式进行的。

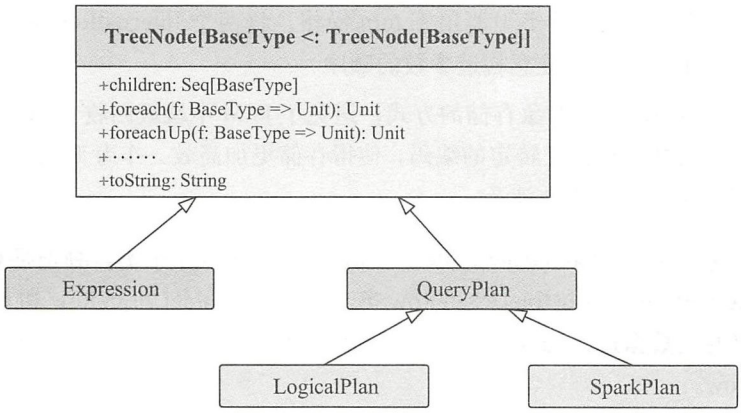


图 3.5　TreeNode 体系

TreeNode 提供的仅仅是一种泛型，实际上包含了两个子类继承体系，即图 3.5 中的 QueryPlan 和 Expression 体系。Expression 是 Catalyst 中的表达式体系，下一节会展开介绍。QueryPlan 类下面又包含逻辑算子树（LogicalPlan）和物理执行算子树（SparkPlan）两个重要的子类，其中逻辑算子树在 Catalyst 中内置实现，可以剥离出来直接应用到其他系统中；而物理算子树 SparkPlan 和 Spark 执行层紧密相关，当 Catalyst 应用到其他计算模型时，可以进行相应的适配修改。

作为基础类，TreeNode 本身仅提供了最简单和最基本的操作。图 3.6 列举了 TreeNode 中现有的一些方法，例如不同遍历方式的 transform 系列方法、用于替换新的子节点的 withNewChildren 方法等。此外，treeString 函数能够将 TreeNode 以树型结构展示，在查看表达式、逻辑算子树和物理算子树时经常用到。

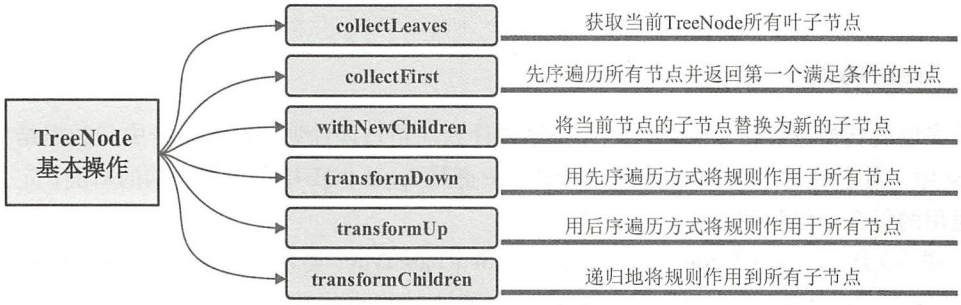


图 3.6　TreeNode 基本操作

除上述操作外，Catalyst 中还提供了节点位置功能，即能够根据 TreeNode 定位到对应的 SQL 字符串中的行数和起始位置。该功能在 SQL 解析发生异常时能够方便用户迅速找到出错的地方，具体参见如下代码。

```
case class Origin(line: Option[Int] = None, startPosition: Option[Int] = None)
object CurrentOrigin {
  private val value = new ThreadLocal[Origin]() {
    override def initialValue: Origin = Origin()
  }
  def get: Origin = value.get()
  def set(o: Origin): Unit = value.set(o)
  def reset(): Unit = value.set(Origin())
  def setPosition(line: Int, start: Int): Unit = {
    value.set(value.get.copy(line = Some(line), startPosition = Some(start)))
  }
  def withOrigin[A](o: Origin)(f: => A): A = {
    set(o)
    val ret = try f finally { reset() }
    reset()
    ret
  }
}
```

可以看到，Origin 提供了 line 和 startPosition 两个构造参数，分别代表行号和偏移量。在 CurrentOrigin 对象中，提供了各种 set 和 get 操作。其中，比较重要的是 withOrigin 方法，支持在 TreeNode 上执行操作的同时修改当前 origin 信息。

3.2.3 Expression 体系

表达式一般指的是不需要触发执行引擎而能够直接进行计算的单元，例如加减乘除四则运算、逻辑操作、转换操作、过滤操作等。如果说 TreeNode 是“框架”，那么 Expression 就是“灵魂”。在各种 SQL 引擎中，表达式（Expression）都起着重要的作用。

Catalyst 实现了完善的表达式体系，与各种算子（QueryPlan）占据同样的地位。算子执行前通常都会进行“绑定”操作，将表达式与输入的属性对应起来，同时算子也能够调用各种表达式处理相应的逻辑。在 Expression 类中，主要定义了 5 个方面的操作，包括基本属性、核心操作、输入输出、字符串表示和等价性判断，如图 3.7 所示。

核心操作中的 eval 函数实现了表达式对应的处理逻辑，也是其他模块调用该表达式的主要接口，而 genCode 和 doGenCode 用于生成表达式对应的 Java 代码（这部分内容将在第 9 章中介绍）。字符串表示用于查看该 Expression 的具体内容，如表达式名和输入参数等。下面对 Expression 包含的基本属性和操作进行简单介绍。

- **foldable**：该属性用来标记表达式能否在查询执行之前直接静态计算。目前，foldable 为 true 的情况有两种，第一种是该表达式为 Literal 类型（“字面值”，例如常量等），第二种是当且仅当其子表达式中 foldable 都为 true 时。当 foldable 为 true 时，在算子树中，表达式可以预先直接处理（“折叠”）。

- **deterministic**：该属性用来标记表达式是否为确定性的，即每次执行 eval 函数的输出是否都相同。考虑到 Spark 分布式执行环境中数据的 Shuffle 操作带来的不确定性，以及某些表达式（如 Rand 等）本身具有不确定性，该属性对于算子树优化中判断谓词能否下推等很有必要。
- **nullable**：该属性用来标记表达式是否可能输出 Null 值，一般在生成的 Java 代码中对相关条件进行判断。
- **references**：返回值为 AttributeSet 类型，表示该 Expression 中会涉及的属性值，默认情况为所有子节点中属性值的集合。
- **canonicalized**：返回经过规范化（Canonicalize）处理后的表达式。规范化处理会在确保输出结果相同的前提下通过一些规则对表达式进行重写，具体逻辑可以参见 Canonicalize 工具类。
- **semanticEquals**：判断两个表达式在语义上是否等价。基本的判断条件是两个表达式都是确定性的（deterministic 为 true）且两个表达式经过规范化处理后（Canonicalized）仍然相同。

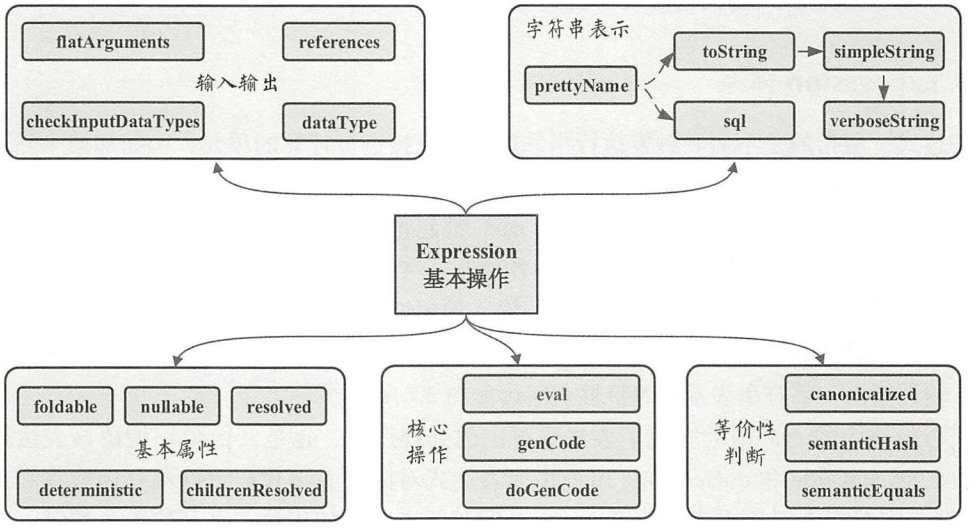


图 3.7 Expression 基本操作

在 Spark SQL 中，Expression 本身也是 TreeNode 类的子类，因此能够调用所有 TreeNode 的方法，例如 transform 等，也可以通过多级的子 Expression 组合成复杂的 Expression。Expression 涉及范围广且数目庞大，相关的类或接口将近 300 个（如图 3.8 所示），这里列举一些比较常用的 Expression 来介绍。

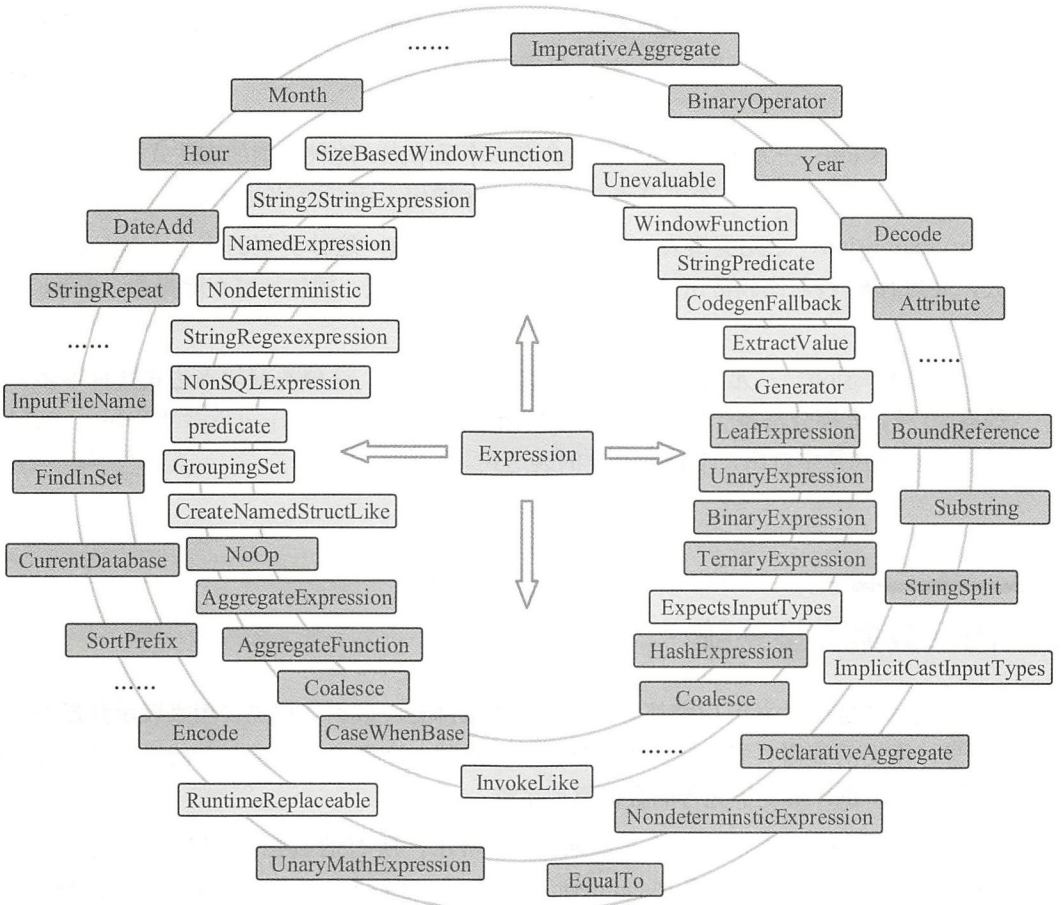


图 3.8 Expression 体系

- Nondeterministic 接口：具有不确定性的 Expression，其中 deterministic 和 foldable 属性都默认返回 false，典型的实现包括 MonotonicallyIncreasingID 表达式、Rand 和 Randn 表达式等。
- Unevaluable 接口：非可执行的表达式，即调用其 eval 函数会抛出异常。该接口主要用于生命周期不超过逻辑计划解析和优化阶段的表达式，例如 Star(*) 表达式在解析阶段就会被展开成具体的列集合。
- CodegenFallback 接口：不支持代码生成的表达式。某些表达式涉及第三方实现（例如 Hive 的 UDF）等情况，无法生成 Java 代码，此时通过 CodegenFallback 直接调用，该接口中实现了具体的调用方法。

- LeafExpression：叶子节点类型的表达式，即不包含任何子节点，因此其 children 方法通常默认返回 Nil 值。该类型的 Expression 目前大约有 30 个，包括 Star、CurrentDate、Pi 表达式等。
- UnaryExpression：一元类型表达式，只含有一个子节点。这种类型的表达式总量 110 多种，较为庞大。其输入涉及一个子节点，例如，Abs 操作、UpCast 表达式等。
- BinaryExpression：二元类型表达式，包含两个子节点。这种类型的表达式数目也比较庞大，大约有 80 种。比较常用的是一些二元的算数表达式，例如加减乘除操作、RLike 函数等。
- TernaryExpression：三元类型表达式，包含 3 个子节点。这种类型的表达式数目不多，大约有 10 种，大部分都是一些字符串操作的函数，非常典型的例子可以参考 Substring 函数，其子节点分别是字符串、下标和长度的表达式。

3.3 内部数据类型系统

数据类型系统是任何 SQL 引擎都必不可少的组成部分。数据类型主要用来表示数据表中存储的列信息，常见的数据类型包括简单的整数、浮点数、字符串，以及复杂的嵌套结构等。在 Spark SQL 中，Catalyst 实现了完善的数据类型系统。

数据类型系统中类的相互继承关系如图 3.9 所示，所有的数据类型都继承自 AbstractDataType 抽象类。比较常用的是各种 NumericType 类型，包括 ByteType（表示一字节的整数，范围是 −128 ～ 127）、ShortType（表示两字节的整数，范围是 −32768 ～ 32767）、IntegerType（表示 4 字节的整数）、LongType（表示 8 字节的整数）、FloatType（表示 4 字节的单精度浮点数）和 DoubleType（表示 8 字节的双精度浮点数）等。另外，DecimalType 是需要特别注意的数据类型，可以用来表示不可变的任意精度的十进制数字，依托内部的 java.math.BigDecimal，支持常规的四则运算和 UDF（例如 round 和 floor 等）。使用 DecimalType 进行转换操作时需要注意 precision 和 scale 值的选取，其中 scale 表示小数部分位数，precision-scale 表示整数部分的位数。

常用的复合数据类型有数组类型（ArrayType）、字典类型（MapType）和结构体类型（StructType）3 种。其中，数组类型（ArrayType）中要求数组元素类型一致；字典类型（MapType）中既要求所有 key 的类型一致，也要求所有的 value 类型一致。

综上所述，Spark SQL 中提供了丰富的数据类型。在实际应用中，可以根据具体的应用需求来选取合适的数据类型（基本类型、复合类型），特别需要注意各种数据类型表示的范围和数据溢出情况的处理。

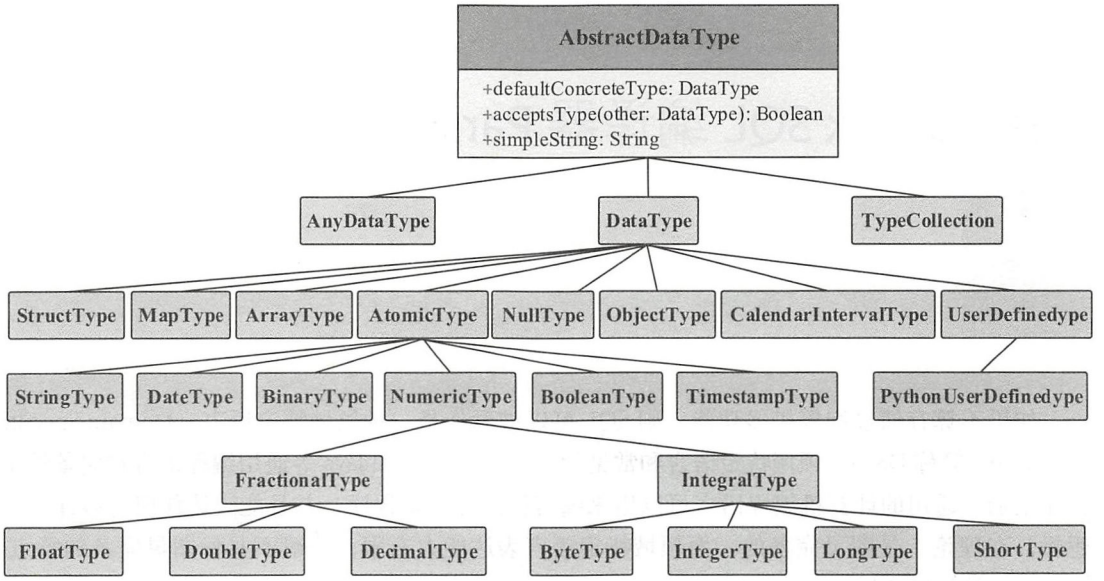


图 3.9 数据类型系统

3.4 本章小结

本章通过一个简单的案例对 SQL 从解析到最终生成 RDD 的过程进行了简要分析，主要包括逻辑计划和物理计划两个重要的阶段，旨在从宏观层面提供大纲视角；同时，也对 Spark SQL 中的 InternalRow、TreeNode 和 Expression 这 3 个重要概念进行了介绍；最后简单列举了 Spark SQL 中定义的内部数据类型。

第4章 Spark SQL 编译器 Parser

如果不算存储过程等扩展功能，则 SQL 可以被看作是一种领域特定语言（Domain Specific Language，简称 DSL）。领域特定语言和常见的 C++、Python 和 Java 等通用编程语言在完备性上存在差别。通用的计算机编程语言可以用来编写任意计算机程序，并且能表达任何可被计算的逻辑，在理论上是图灵完备的。而领域特定语言表达能力有限，一般不具备图灵完备的特点，可在特定的领域或场景解决特定的任务。

相对于形态固定的 API，DSL 提供了更为灵活和强大的用户可编程接口，因此在实际项目中，DSL 的应用非常多。例如，对于正则表达式中字符串模式（Pattern）的语法，开发人员仅需要编写相应的 Pattern，匹配引擎就会根据其逻辑处理字符串。此外，Web 开发中的 HTML 和 CSS 语法，也可以看作是 Web 的领域特定语言。

4.1 DSL 工具之 ANTLR 简介

DSL 的构建与通用编程语言的构建类似，主要的过程仍然是指定语法和语义，然后实现编译器或解释器。通常情况下，一个系统中 DSL 模块的实现需要涉及两方面的工作。

- 设计语法和语义，定义 DSL 中具体的元素。
- 实现词法分析器（Lexer）和语法分析器（Parser），完成对 DSL 的解析，最终转换为底层逻辑来执行。

经过几十年的研究，编译理论已经比较成熟，借助于各种工具，开发人员不再需要从头开始构建烦琐的词法分析和语法分析模块。迄今为止，业界提供了各种各样的生成器（Generator），可以直接应用在系统中。例如，在 C/C++ 开发环境下比较有名的 Flex、Yacc 等；在 Java 的世界里，词法分析工具有 JFlex，语法分析工具有 BYACC/J、Java Cup，兼具两者的有 JavaCC 和 ANTLR 等。基于生成器，实现一个编译器前端就像“填表”那么简单，只要提供特定的文法即可。

ANTLR（Another Tool for Language Recognition）是目前非常活跃的语法生成工具，用 Java 语言编写，基于 LL（*）解析方式[28]，使用自上而下的递归下降分析方法。ANTLR 可以用来产生词法分析器、语法分析器和树状分析器（Tree Parser）等各个模块，其文法定义使用类似 EBNF（Extended Backus-Naur Form）的方式，简洁直观。ANTLR 本身使用 switch-case 逻辑来匹配字符（Token），形成记号序列流。从 20 世纪 80 年代末发展至今，ANTLR 已经升级到 ANTLR 4，并形成了较为成熟的生态链。

类似于所有的 LL 解析器，早期的 ANTLR 也无法有效处理“左递归”（Left-recursion），后来在 ANTLR 3 中将左递归的解析策略退化为 LL（1）加上回溯的形式。ANTLR 4 支持 Kleene Closure 表示法，以 while 语句取代递归来解决 LL 分析法所不允许的左递归。对于 LL（*）不能正确分析的特殊情况，ANTLR 4 还支持语义断言（Semantic Predicate）来辅助判断。同时，ANTLR 4 使用新的 ALL(*) 算法（Adaptive LL(*)），能够在运行时动态分析语法，在解决歧义与分支决策的时候更加高效智能。

ANTLR 4 除能够自动构建语法分析树外，还支持生成基于监听器（Listener）模式和访问者（Visitor）模式的树遍历器。访问者模式遍历语法树是一种更加灵活的方式，可以避免在文法文件中嵌入烦琐的动作（Action），使解析与应用代码分离，这样不但文法的定义更加简洁清晰，而且可以在不重新编译生成语法分析器的情况下复用相同的语法，甚至能够采用不同的程序语言来实现这些动作。

ANTLR 的应用非常广泛，Hibernate 与 WebLogic 都使用 ANTLR 解析 HQL 语言，NetBeans IDE 中基于 ANTLR 解析 C++，Twitter 搜索模块依赖于 ANTLR，Hive、Presto 和 Spark SQL 等大数据引擎的 SQL 编译模块也都是基于 ANTLR 构建的。

4.1.1 基于 ANTLR 4 的计算器

ANTLR 4 是进行 Spark SQL 开发的基础，本小节以简单的四则运算为例，使用 ANTLR 4 构建计算器。在 ANTLR 4 中，词法和语法可以放在同一个 G4 文件中，词法单元以大写字母开头，语法单元以小写字母开头，以作区分。

```
grammar Calculator;

line : expr EOF ;
expr : '(' expr ')'            # parenExpr
     | expr ('*'|'/') expr     # multOrDiv
     | expr ('+'|'-') expr     # addOrSubtract
     | FLOAT                   # float

WS : [ \t\n\r]+ -> skip ;
FLOAT : DIGIT+ '.' DIGIT* EXPONENT?
      | '.' DIGIT+ EXPONENT?
```

```
    | DIGIT+ EXPONENT? ;
fragment DIGIT : '0'..'9';
fragment EXPONENT : ('e'|'E') ('+'|'-')? DIGIT+ ;
```

在上述文法文件中，“grammar Calculator”表示文件是一个词法、语法混合文件，名称必须和文件名相同，即该文法文件的文件名应该是 Calculator.g4 。文法规则比较简单，仅支持加减乘除和加括号的写法。需要注意的是，expr 每条规则后面的“#”是产生式标签名（Alternative Label Name），起到标记不同规则的作用。词法单元 DIGIT 前的 fragment 表示这是个词片段，不会生成对应的 Token。“'0'..'9'”表示 0 ～ 9 的字符，和 [0-9] 的意义一样。词法规则 WS 定义了空格，其中的“-> skip”是 ANTLR 4 中特殊的命令，表示直接跳过不做任何处理。整个文法是与目标语言无关的，同样的文件可以生成 Java、JavaScript、Python 等不同语言的代码。

基于 Calculator 文法文件，可以直接在命令行下或 MAVEN 中调用 ANTLR 4 生成相应的代码，IDEA 等集成开发环境也提供了 ANTLR 4 的插件支持，具体的操作步骤可以参考官方网站相关资料，这里不再赘述。完整的生成代码如图 4.1 所示，其中 Calculator.tokens 和 CalculatorLexer.tokens 是内部的 Token 定义，CalculatorLexer 和 CalculatorParser 是生成的词法分析器和语法分析器。剩下的 Java 文件代表着两种访问语法树的方式，CalculatoListener 和 CalculatoBaseListener 对应监听器模式，CalculatoVisitor 和 CalculatoBaseVisitor 对应访问者模式。

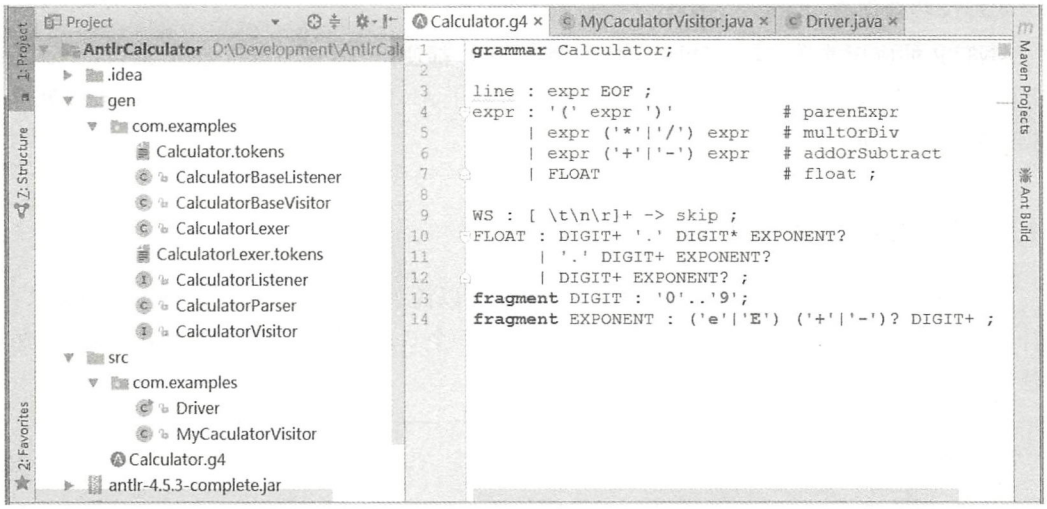


图 4.1　ANTLR 4 文法定义与代码生成

基于生成的代码，开发人员只要实现语法树遍历过程中的核心逻辑即可，可以在监听器模式和访问者模式中任意选择。考虑到 Spark SQL 编译器中主要采用 Visitor 方式，这里在 CalculatorBaseVisitor 的基础上继承自己的类，重载其中的关键方法。代码片段如下所示，可以看到文法文件中的 addOrSubtract 标签对应于 visitAddOrSubstract 方法，在其中实现加减法的逻

辑即可，而 visitFloat 方法和 float 标签一一对应，完成浮点数的解析。

```
public class MyCaculatorVisitor extends CalculatorBaseVisitor<Object> {
    ..........................................................
    @Override
    public Object visitAddOrSubtract(CalculatorParser.AddOrSubtractContext ctx) {
        Object obj0 = ctx.expr(0).accept(this);
        Object obj1 = ctx.expr(1).accept(this);
        if ("+".equals(ctx.getChild(1).getText())) {
            return (Float)obj0 + (Float)obj1;
        } else if ("-".equals(ctx.getChild(1).getText())) {
            return (Float)obj0 - (Float)obj1;
        }
        return 0f;
    }

    @Override
    public Object visitFloat(CalculatorParser.FloatContext ctx) {
        return Float.parseFloat(ctx.getText());
    }
}
```

实现了 Visitor 中的关键逻辑以后，就可以直接调用 ANTLR 4 生成的各个模块了，驱动程序如下，根据输入的字符流相继构造词法分析器（Lexer）和语法分析器（Parser），然后创建相应的 Visitor 来访问语法分析器解析得到的语法树，最后返回结果。

```
public class Driver {
    public static void main(String[] args) {
        String query = "3.1*(6.3-4.51)";
        CalculatorLexer lexer = new CalculatorLexer(new ANTLRInputStream(query));
        CalculatorParser parser = new CalculatorParser(new CommonTokenStream(lexer));
        CalculatorVisitor visitor = new MyCaculatorVisitor();
        System.out.println(visitor.visit(parser.expr()));
    }
}
```

相比于上述例子，实际系统中的文法规则更加复杂，开发过程中也会涉及更多的编译原理与 ANTLR 本身的知识，有兴趣的读者可以进一步参考相关资料 [29]。

4.1.2 访问者模式

访问者模式是一种将算法与对象结构分离的软件设计模式。这种模式的工作方法如下：假设拥有一个由许多对象构成的对象结构，这些对象的类都拥有一个 accept 方法用来接受访问者对象；访问者是一个接口，它拥有一个 visit 方法，这个方法对访问到的对象结构中不同类型的元素做出不同的反应；在对象结构的一次访问过程中，遍历整个对象结构，对每个元素都实施

accept 方法，在每个元素的 accept 方法中回调访问者的 visit 方法，从而使访问者得以处理对象结构的每个元素；可以针对对象结构设计不同的具体访问者类来完成不同的操作。

这里通过一个实际场景来介绍访问者模式：假设某旅游爱好者决定在假期去北京、上海和深圳旅游，在每个城市都会进行一些不同的活动，需要通过访问者模式来实现。

总的设计方案如图 4.2 所示。首先，针对不同城市抽象出一个 City 接口。每个城市都通过 accept 方法接受访问者，其中 Visitor 表示访问人员的接口。

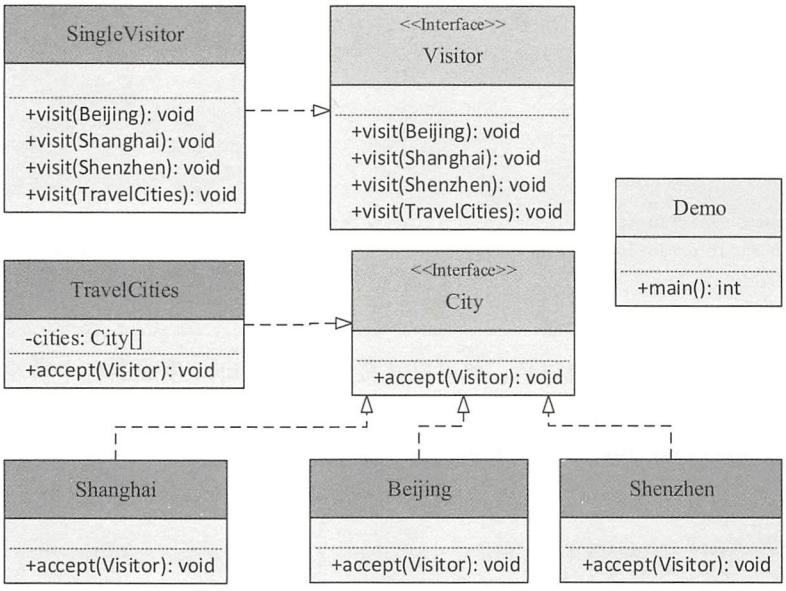


图 4.2　访问者模式实例

```
public interface City {
  public void accept(Visitor visitor);
}
```

接着，创建具体的城市实体类，包括上述 3 个城市。每个城市的 accept 实现完全相同，调用访问者接口中实现的 visit 方法进行对应的活动。

```
public class Beijing implements City {
   @Override
   public void accept(Visitor visitor) {
      visitor.visit(this);
   }
}
```

```
public class Shanghai implements City {
   @Override
   public void accept(Visitor visitor) {
      visitor.visit(this);
   }
}
```

```
public class Shenzhen implements City {
   @Override
   public void accept(Visitor visitor) {
      visitor.visit(this);
   }
}
```

将上述 3 个城市整合在旅游计划中。此外，用 TravelCities 类实现 City 的接口，在 accept 中对初始化构造的城市列表依次进行访问。

```
public class TravelCities implements City {
   City[] cities;
   public TravelCities {
      cities = new City[] {new Beijing(), new Shanghai(), new Shenzhen()};
   }
   @Override
   public void accept(Visitor visitor) {
      for (int i = 0; i < cities.length; i++) {
         cities[i].accept(visitor);
      }
   }
}
```

然后，定义访问者 Visitor 接口。需要注意，在该接口中包含访问所有城市的 visit 方法，分别代表该访问者在不同城市的具体活动。可以看到，这里的 Visitor 和 4.1.1 小节的 CalculatorVisitor 接口相似。

```
public interface Visitor {
    public void visit(Beijing beijing);
    public void visit(Shanghai shanghai);
    public void visit(Shenzhen shenzhen);
}
```

假设旅行者是一个人出行，这里实现一个 SingleVisitor 接口，代表该旅行者在不同城市所进行的活动。

```
public interface SingleVisitor implements Visitor {
   @Override
```

```
    public void visit(Beijing beijing) {
        System.out.println( "Hello Beijing!" );
    }
    @Override
    public void visit(Shanghai shanghai) {
        System.out.println( "Hello Shanghai!" );
    }
    @Override
    public void visit(Shenzhen shenzhen) {
        System.out.println( "Hello Shenzhen!" );
    }
}
```

最后，通过一个 Demo 类来执行整个流程。

```
public class Demo {
    public static void main(String[] args) {
        TravelCities travelcities = new TravelCities();
        travelcities.accept(new SingleVisitor());
    }
}
```

上述案例使用访问者模式把结构和行为分离，如果需要多增加行为，则添加新的 Visitor 或重载已有的 Visitor 并修改特定的访问行为。读者在后续章节中会看到，在 Spark SQL 的编译器后端实现逻辑中，有很多类似的情形。

4.2 SparkSqlParser 之 AstBuilder

回到 Spark SQL，Catalyst 中提供了直接面向用户的 ParseInterface 接口，该接口中包含了对 SQL 语句、Expression 表达式和 TableIdentifier 数据表标识符的解析方法。AbstractSqlParser 是实现了 ParseInterface 的虚类，其中定义了返回 AstBuilder 的函数。

整个 SQL 解析相关的实现如图 4.3 所示，其中 CatalystSqlParser 仅用于 Catalyst 内部，而 SparkSqlParser 用于外部调用。其中，比较核心的是 AstBuilder，它继承了 ANTLR 4 生成的默认 SqlBaseBaseVisitor，用于生成 SQL 对应的抽象语法树 AST（Unresolved LogicalPlan）；SparkSqlAstBuilder 继承 AstBuilder，并在其基础上定义了一些 DDL 语句的访问操作，主要在 SparkSqlParser 中调用。

当面临开发新的语法支持时，首先需要改动的是 ANTLR 4 文件（在 SqlBase.g4 中添加文法），重新生成词法分析器（SqlBaseLexer）、语法分析器（SqlBaseParser）和访问者类（SqlBaseVisitor 接口与 SqlBaseBaseVisitor 类），然后在 AstBuilder 等类中添加相应的访问逻辑，最后添加执行逻辑。

```
lazy val sqlParser: ParserInterface = new SparkSqlParser(conf)
```

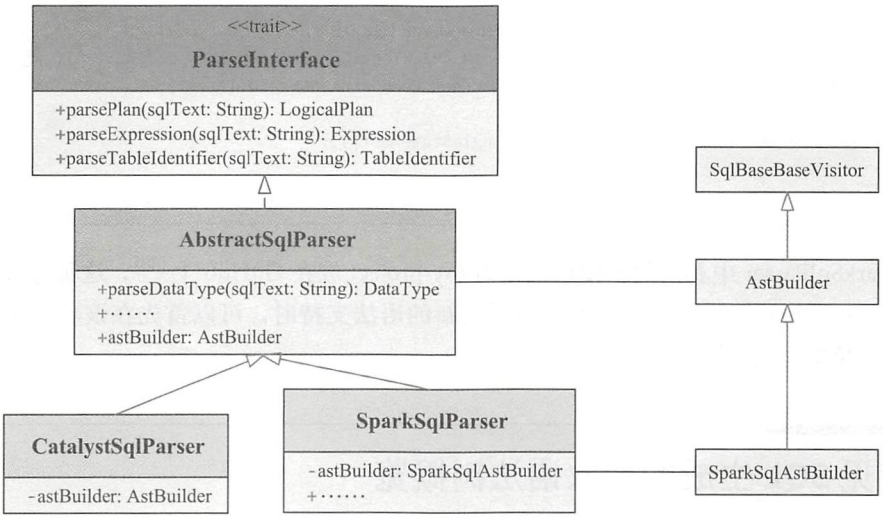


图 4.3 Spark SQL 编译器

为加深理解 Spark SQL 生成的语法树结构，读者可以将 Spark SQL 编译器部分剥离出来，构造一个类似 AstBuilder 的访问者类 MyVisitor，在实现的访问方法中输出 visitor 访问操作。类似于下面的代码逻辑，实现 SqlBaseBaseVisitor 中的所有方法。

```
public class MyVisitor extends SqlBaseBaseVisitor<String>{
    public String visitSingleStatement(SqlBaseParser.SingleStatementContext ctx) {
        System.out.println( “visitSingleStatement” );
        return visitChildren(ctx);
    }
    ..........................................................................
}
```

Note：需要注意的是，上述代码中的访问操作顺序统一为先序遍历。而 AstBuilder 中的操作则遵循后序遍历方式（首先会生成子节点的 LogicalPlan，然后生成当前节点的 LogicalPlan）。AstBuilder 左右子节点的访问顺序也不固定，读者可以对上述代码中的节点访问顺序进行相应调整。

MyVisitor 中访问方法的类型为 String（AstBuilder 中的 SqlBaseBaseVisitor 为 AnyRef 类型，返回 LogicalPlan 类型），但不会返回字符串，仅用于输出访问的路径和对 AST 的理解。构造上述访问者类之后，接下来还需要构造一个 Driver 程序来驱动上述访问过程。ParserDriver 的代码如下。

```
public class ParserDriver {
    public static void main(String[] args) {
        String query = "select name from studeng where age>18" ;
        SqlBaseLexer lexer = new SqlBaseLexer(new ANTLRInputStream(query.toUpperCase()));
        SqlBaseParser parser = new SqlBaseParser(new CommonTokenStream(lexer));
        MyVisitor visitor = new MyVisitor();
        visitor.visitSingleStatement(parser.singleStatement());
    }
}
```

从 SparkSqlParser 中剥离的子模块已作为 toy-project 放在 GitHub 上[30]，这种方式有助于读者理解 Spark SQL 的解析过程。此外，当开发新的语法支持时，可以首先在该项目上进行初步的验证，避免编译 Spark SQL。

4.3 常见 SQL 生成的抽象语法树概览

本节将常用的一些查询语句可视化，帮助读者对 SQL 编译器进行理解。在 Catalyst 中，SQL 语句经过解析，生成的抽象语法树节点都以 Context 结尾来命名。如图 4.4 所示为第 3 章案例中的 SQL 查询语句生成的抽象语法树。

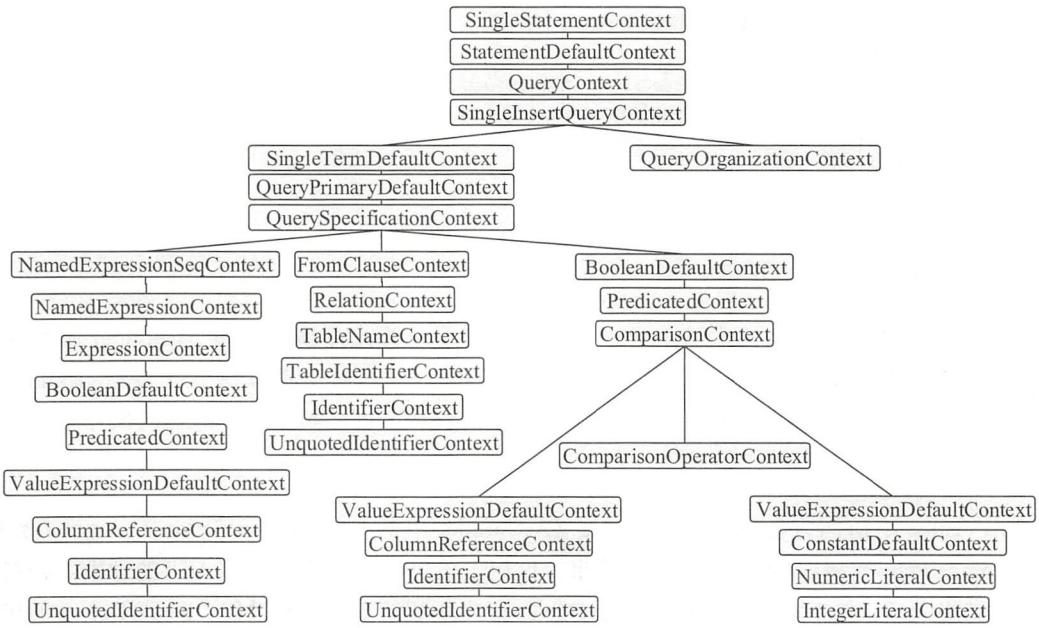


图 4.4　案例对应的抽象语法树

从语法树可以看到，SingleStatementContext 是根节点，但是在访问该节点时一般什么都不做，只递归访问子节点。整个遍历访问操作中比较重要的是包含多个子节点的节点。例如 QuerySpecificationContext 节点，一般将数据表和具体的查询表达式整合在一起。左边的一系列节点对应 select 表达式中选择的列，中间的 FromClauseContext 为根节点的系列节点对应数据表，右边的一系列节点则对应 where 条件中的表达式。

上述语法树的结构比较通用，其他类型的 SQL 语句生成的语法树大同小异，这里假设在上述语句中加入排序的操作。

```
select name from student where age > 18 order by id desc
```

加入排序操作后生成的语法树如图 4.5 所示，可以看到新的语法树在 QueryOrganization-Context 节点下面加入了 SortItemContext 节点，代表数据查询之后所进行的排序操作。一般来讲，QueryOrganizationContext 为根节点所代表的子树中包含了各种对数据组织的操作，例如 Sort、Limit 和 Window 算子等。

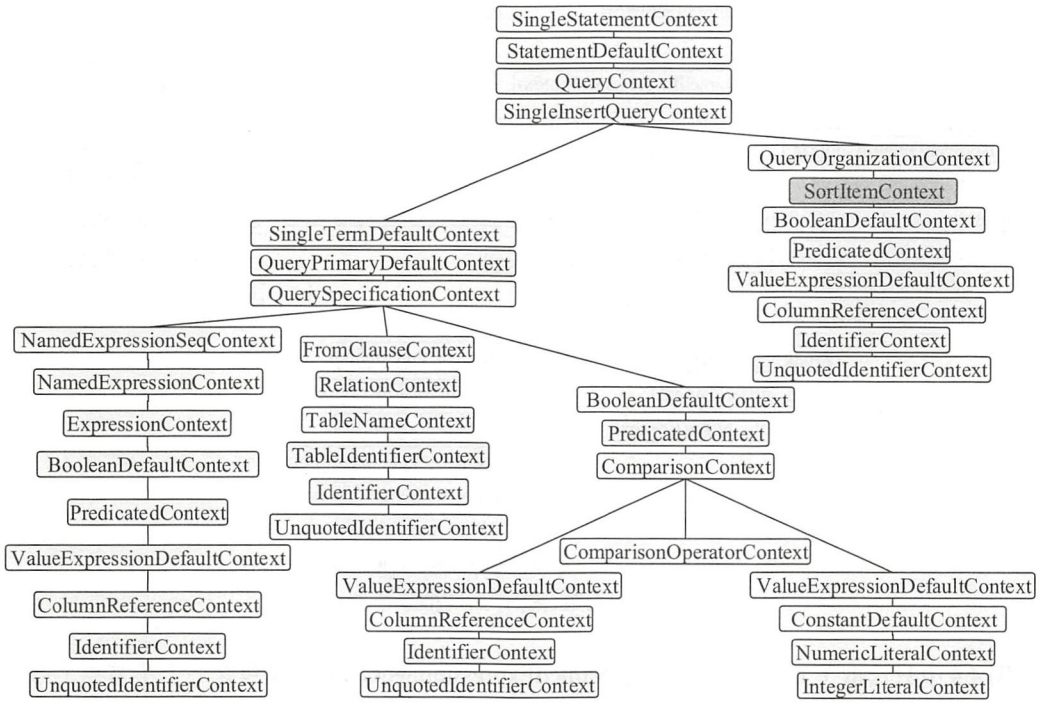


图 4.5 加入排序操作后的抽象语法树

从上面案例可以看出，即使非常简单的 SQL 语句，其语法树的节点也非常多。特别是当查询涉及聚合操作、Join 操作和嵌套的子查询时，整棵语法树会变得非常庞大，难以一次完成

展示。本节最后再讲解一下包含聚合操作的 SQL 语句生成的语法树。同样地，查询针对的是 student 数据表，将其中的 id 列进行分组（group by）操作，每组的聚合函数使用 count 统计总数。

```
select id, count(name) from student group by id
```

加入聚合操作后生成的语法树如图 4.6 所示，将其与图 4.4 中展示的语法树进行对比。可以看到，根节点周围的结构没有改变，变化的主要是 QuerySpecificationContext 节点所包含的子树。

图 4.6　加入聚合操作后的抽象语法树

图 4.4 中只选取了 name 一列，因此 NamedExpressionSeqContext 仅包含一个子节点（NamedExpressionContext）。在上述聚合查询中，除 id 外，还有对 name 的 count 操作所产生的新列，因此 NamedExpressionSeqContext 节点包含两个子节点。FromClauseContext 子树代表的数据表信息没有变化，仍然是 QuerySpecificationContext 的第二个子节点。加入聚合操作后的语法树最重要的元素是 FunctionCallContext 节点和 AggregationContext 节点。在 SqlBase.g4 文法文件中表示

聚合操作的关键字是 group by、cube、grouping sets 和 roll up 这 4 种。反映在语法树中就是图 4.6 中 QuerySpecificationContext 节点下的 AggregationContext 节点。表示聚合函数的 FunctionCallContext 节点很好理解，其子节点 QualifiedNameContext 代表函数名，ExpressionContext 表示函数的参数表达式(对应 SQL 语句中的 name 列)。

4.4 本章小结

查询语句的翻译是数据分析的第一步，本章重点讲解了 Spark SQL 中的 SQL 编译器，并对其中涉及的基本原理与设计模式进行了介绍。最后，结合具体案例对常见语句的语法树进行了可视化展示以加深理解，为学习后续章节做准备。

第5章 Spark SQL 逻辑计划（LogicalPlan）

逻辑计划阶段在整个流程中起着承前启后的作用。在此阶段，字符串形态的 SQL 语句转换为树结构形态的逻辑算子树，SQL 中所包含的各种处理逻辑（过滤、剪裁等）和数据信息都会被整合在逻辑算子树的不同节点中。逻辑计划本质上是一种中间过程表示，与 Spark 平台无关，后续阶段会进一步将其映射为可执行的物理计划。

5.1 Spark SQL 逻辑计划概述

Spark SQL 逻辑计划在实现层面被定义为 LogicalPlan 类。从 SQL 语句经过 SparkSqlParser 解析生成 Unresolved LogicalPlan，到最终优化成为 Optimized LogicalPlan，这个流程主要经过 3 个阶段，如图 5.1 所示。这 3 个阶段分别产生 Unresolved LogicalPlan、Analyzed LogicalPlan 和 Optimized LogicalPlan，其中 Optimized LogicalPlan 传递到下一个阶段用于物理执行计划的生成。

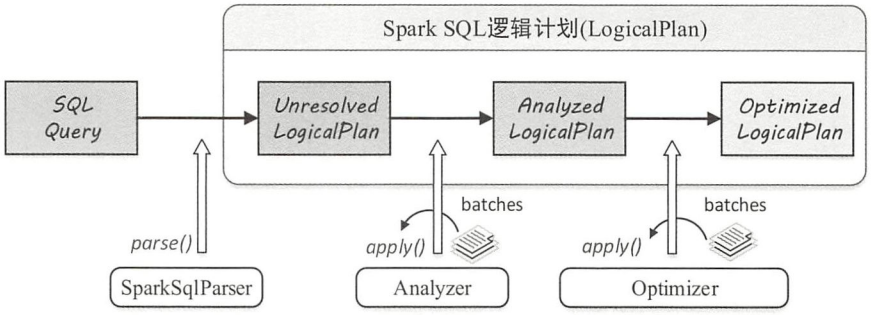


图 5.1　逻辑计划的 3 个阶段

具体来讲，这 3 个阶段所完成的工作分别如下。

（1）由 SparkSqlParser 中的 AstBuilder 执行节点访问，将语法树的各种 Context 节点转换成对应的 LogicalPlan 节点，从而成为一棵未解析的逻辑算子树（Unresolved LogicalPlan），此时的

逻辑算子树是最初形态，不包含数据信息与列信息等。

（2）由 Analyzer 将一系列的规则作用在 Unresolved LogicalPlan 上，对树上的节点绑定各种数据信息，生成解析后的逻辑算子树（Analyzed LogicalPlan）。

（3）由 Spark SQL 中的优化器（Optimizer）将一系列优化规则作用到上一步生成的逻辑算子树中，在确保结果正确的前提下改写其中的低效结构，生成优化后的逻辑算子树（Optimized LogicalPlan）。

本章首先在 5.2 节全面介绍 LogicalPlan 所涉及的方方面面的基础知识，包括 LogicalPlan 分类和各种操作的概述；然后，在 5.3～5.5 节分别讲解 3 个阶段的执行过程；最后，在 5.6 节对全章内容进行总结。

5.2 LogicalPlan 简介

LogicalPlan 作为数据结构记录了对应逻辑算子树节点的基本信息和基本操作，包括输入输出和各种处理逻辑等。在第 3 章中已经介绍过，LogicalPlan 属于 TreeNode 体系，继承自 QueryPlan 父类。

5.2.1 QueryPlan 概述

在介绍 LogicalPlan 之前，有必要先介绍一下其父类 QueryPlan。为了从整体视角了解 QueryPlan 的所有功能，图 5.2 列举了其全部操作，并进行了简单的类型划分。

图 5.2 将 QueryPlan 的主要操作分为 6 个模块，分别是输入输出、字符串、规范化、表达式操作、基本属性和约束。下面简单介绍这 6 个模块，读者可结合实际代码阅读。

- 输入输出：QueryPlan 的输入输出定义了 5 个方法，其中 output 是返回值为 Seq[Attribute] 的虚函数，具体内容由不同子节点实现，而 outputSet 是将 output 的返回值进行封装，得到 AttributeSet 集合类型的结果。获取输入属性的方法 inputSet 的返回值也是 AttributeSet，节点的输入属性对应所有子节点的输出；producedAttributes 表示该节点所产生的属性；missingInput 表示该节点表达式中涉及的但是其子节点输出中并不包含的属性。
- 基本属性：表示 QueryPlan 节点中的一些基本信息，其中 schema 对应 output 输出属性的 schema 信息，allAttributes 记录节点所涉及的所有属性（Attribute）列表，aliasMap 记录节点与子节点表达式中所有的别名信息，references 表示节点表达式中所涉及的所有属性集合，subqueries 和 innerChildren 都默认实现该 QueryPlan 节点中包含的所有子查询。
- 字符串：这部分方法主要用于输出打印 QueryPlan 树型结构信息，其中 schema 信息也会以树状展示。需要注意的一个方法是 statePrefix，用来表示节点对应计划状态的前缀字符

串。在 QueryPlan 的默认实现中，如果该计划不可用（invalid），则前缀会用感叹号（“!”）标记。

- 规范化：类似 Expression 中的方法定义，对 QueryPlan 节点类型也有规范化（Canonicalize）的概念。在 QueryPlan 的默认实现中，canonicalized 直接赋值为当前的 QueryPlan 类；此外，在 sameResult 方法中会利用 canonicalized 来判断两个 QueryPlan 的输出数据是否相同。
- 表达式操作：在第 3 章中已经介绍过 Spark SQL 丰富的表达式体系，其典型的特点就是不需要驱动，直接执行。而在 QueryPlan 各个节点中，包含了各种表达式对象，各种逻辑操作一般也都是通过表达式来执行的。在 QueryPlan 的方法定义中，表达式相关的操作占据重要的地位，其中 expressions 方法能够得到该节点中的所有表达式列表，其他方法很容易根据命名了解对应功能，具体的实现细节可以参看代码。
- 约束（Constraints）：本质上也属于数据过滤条件（Filter）的一种，同样是表达式类型。相对于显式的过滤条件，约束信息可以“推导”出来，例如，对于“a > 5”这样的过滤条件，显然 a 的属性不能为 null，这样就可以对应地构造 isNotNull（a）约束；又如“a = 5”和“a = b”的谓词，能够推导得到“b = 5”的约束条件。在实际情况下，SQL 语句中可能会涉及很复杂的约束条件处理，如约束合并、等价性判断等。在 QueryPlan 类中，提供了大量方法用于辅助生成 constraints 表达式集合以支持后续优化操作。例如，validConstraints 方法返回该 QueryPlan 所有可用的约束条件，比较常用的 constructIsNotNullConstraints 方法，会针对特定的列构造 isNotNull 的约束条件。

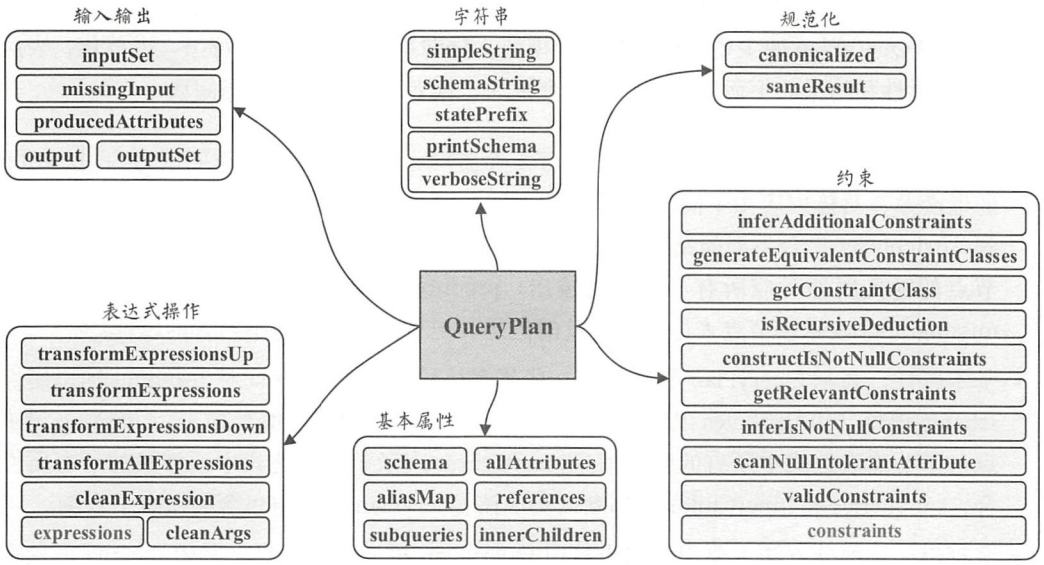


图 5.2　QueryPlan 基本操作

5.2.2 LogicalPlan 基本操作与分类

本小节进行 LogicalPlan 类的分析，介绍逻辑算子树节点都有哪些功能。同样的，在内容上首先从整体列出 LogicalPlan 中的所有方法，使读者对 LogicalPlan 有一个宏观的概念。读者可结合代码阅读。

如图 5.3 所示，LogicalPlan 继承自 QueryPlan，包含了两个成员变量和 17 个方法。两个成员变量一个是 resolved，用来标记该 LogicalPlan 是否为经过了解析的布尔类型值；另一个是 canonicalized，重载了 QueryPlan 中的对应赋值，默认实现消除了子查询别名之后的 LogicalPlan。

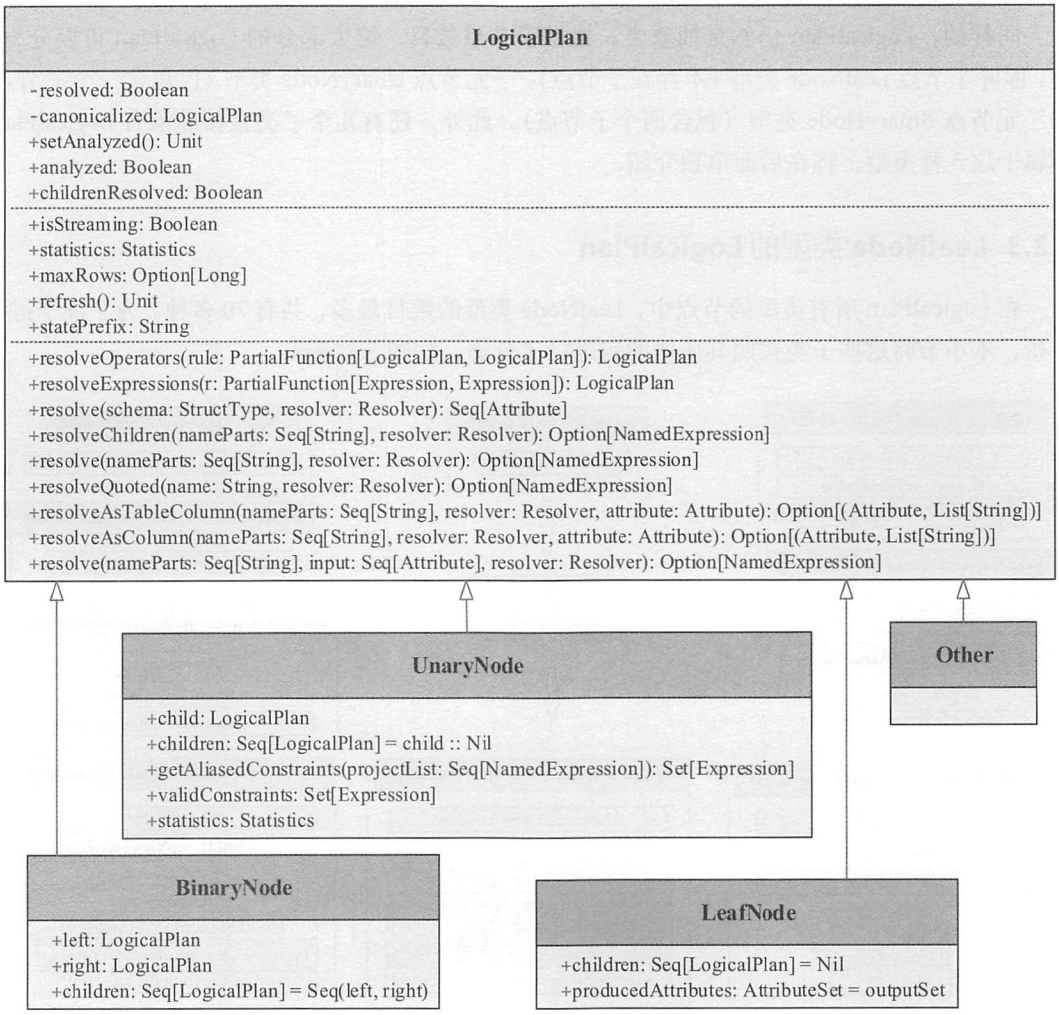


图 5.3 LogicalPlan 基本操作

图 5.3 中的方法根据操作的内容进行了分类，前 3 个方法与 resolved 成员变量相关，其中 childrenResolved 标记子节点是否已经被解析。中间的 5 个方法设定了该 LogicalPlan 中的一些基本信息，其中 statePrefix 重载了 QueryPlan 中的实现，如果该逻辑算子树节点未经过解析，则输出的字符串前缀会加上单引号（'）；isStreaming 方法用来表示当前逻辑算子树中是否包含流式数据源；statistics 记录了当前节点的统计信息，例如默认实现的 sizeInBytes 信息，一般来讲如果当前节点不包含子节点，则必须重载实现该方法；maxRows 记录了当前节点可能计算的最大行数，一般常用于 Limit 算子；refresh 方法会递归地刷新当前计划中的元数据等信息。剩下的则是 LogicalPlan 中定义的与 resolve 相关的 9 个分析方法，用来执行对数据表、表达式、schema 和列属性等类型的解析，具体实现可以参见代码。

同样的，LogicalPlan 仍然是抽象类，根据子节点数目，绝大部分的 LogicalPlan 可以分为 3 类，即叶子节点 LeafNode 类型（不存在子节点）、一元节点 UnaryNode 类型（仅包含一个子节点）和二元节点 BinaryNode 类型（包含两个子节点）。此外，还有几个子类直接继承自 LogicalPlan，不属于这 3 种类型，将在后面单独介绍。

5.2.3 LeafNode 类型的 LogicalPlan

在 LogicalPlan 所有类型的节点中，LeafNode 类型的数目最多，共有 70 多种。为了便于进行分析，本小节将这些子类按照其所属的包进行了分类，如图 5.4 所示。

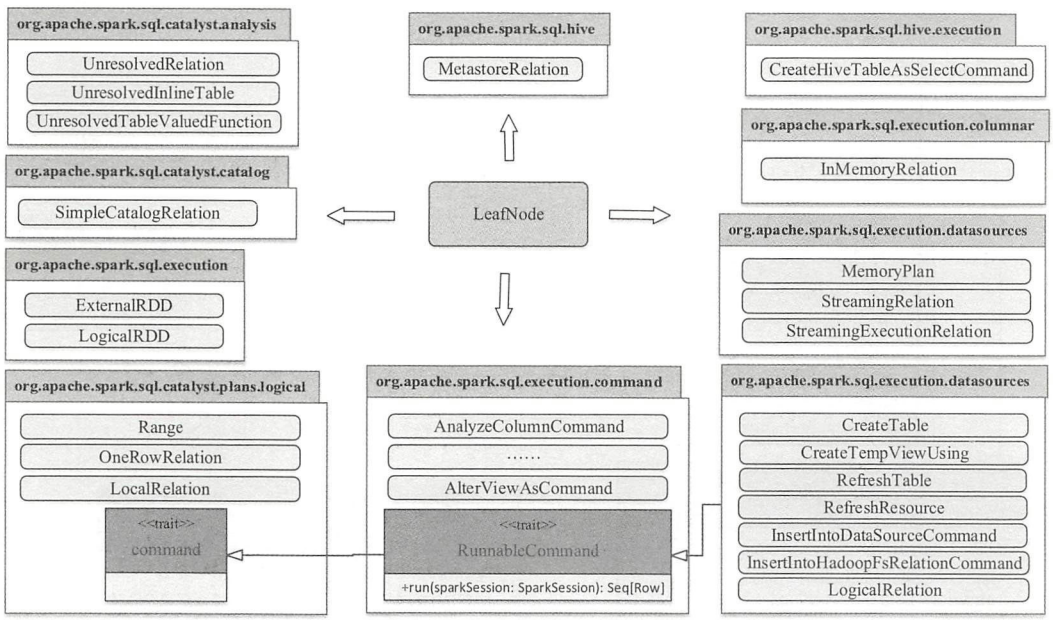


图 5.4　LeafNode 类型的 LogicalPlan

根据图 5.4 中的关键字可以看到，LeafNode 类型的 LogicalPlan 节点对应数据表和命令（Command）相关的逻辑，因此这些 LeafNode 子类中有很大一部分都属于 datasources 包和 command 包。值得一提的是，实现 RunnableCommand 特质（Trait）的类共有 40 多个，是数量最为庞大的 LogicalPlan 类型。顾名思义，RunnableCommand 是直接运行的命令，主要涉及 12 种情形，包括 Database 相关命令、Table 相关命令、View 相关命令、DDL 相关命令、Function 相关命令和 Resource 相关命令等。例如，Database 相关命令有两个，分别是 ShowDatabasesCommand 和 SetDatabaseCommand，用于显示当前数据库名称和切换当前数据库到相应的数据库。

以 SetDatabaseCommand 命令为例，其实现比较简单，如以下代码所示，该 Command 对应 SQL 中的“use database”语句。可以看到，该命令直接调用的是 Catalog 中的 set 方法。Catalog 相关的内容会在 5.4 节中进行介绍。

```
case class SetDatabaseCommand(databaseName: String) extends RunnableCommand {
  override def run(sparkSession: SparkSession): Seq[Row] = {
    sparkSession.sessionState.catalog.setCurrentDatabase(databaseName)
    Seq.empty[Row]
  }
}
```

5.2.4 UnaryNode 类型的 LogicalPlan

在 LogicalPlan 所有类型的节点中，UnaryNode 类型的节点应用非常广泛，共有 34 种，常见于对数据的逻辑转换操作，包括过滤等，如图 5.5 所示。

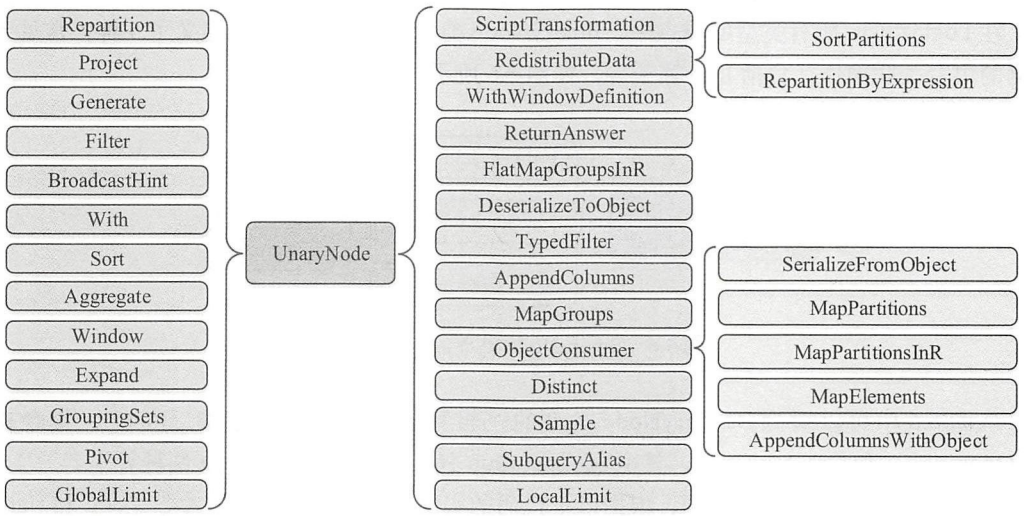


图 5.5 UnaryNode 类型的 LogicalPlan

根据节点所起的不同作用，图 5.5 中的所有节点可以分为 4 个类别。

- 用来定义重分区（repartitioning）操作的 3 个 UnaryNode，即 RedistributeData 及其两个子类 SortPartitions 和 RepartitionByExpression，主要针对现有分区和排序的特点不满足的场景。
- 脚本相关的转换操作 ScriptTransformation，用特定的脚本对输入数据进行转换。
- Object 相关的操作，即 ObjectConsumer 这个特质（Trait）和其他 10 个类，包括 DeserializeToObject、SerializeFromObject 和 FlatMapGroupsInR 等。
- 基本操作算子（basicLogicalOperators），数量最多，共有 19 种，涉及 Project、Filter、Sort 等各种常见的关系算子。

这里以 Sort 节点为例，根据以下代码中的类的定义可以看到，其实现同样非常简单，不涉及额外的逻辑处理；基本上只保存了 Sort 操作中所需要的相关信息，包括排序的规则（升序或降序表达式）、是否全局等。类似的，其他基本操作对应逻辑算子的实现也比较简单，具体可参见代码。

```
case class Sort(order: Seq[SortOrder], global: Boolean, child: LogicalPlan) extends UnaryNode {
  override def output: Seq[Attribute] = child.output
  override def maxRows: Option[Long] = child.maxRows
}
```

5.2.5 BinaryNode 类型的 LogicalPlan

在 LogicalPlan 所有类型的节点中，BinaryNode 类型的节点很少，只定义了 5 种，常见于对数据的组合关联操作，包括 Join 算子等，如图 5.6 所示。

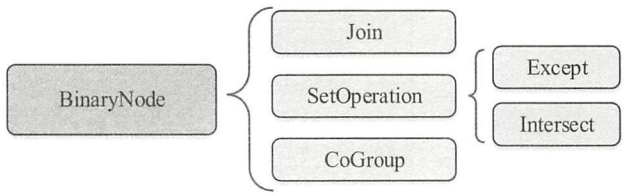


图 5.6　BinaryNode 类型的 LogicalPlan

从图 5.6 中可以看到，BinaryNode 类型的逻辑算子树节点包括连接（Join）、集合操作（SetOperation）和 CoGroup 3 种，其中 SetOperation 包括 Except 和 Intersect 两种算子。

同样的，以 Except 算子为例，根据以下代码的实现可以看到，其传入参数是 left 和 right 两个 LogicalPlan，重载实现的 output 方法、validContraints 方法甚至 statistics 方法都只需直接调用

输出 left 的方法。BinaryNode 类型节点中比较复杂且重要的是 Join 算子，这部分内容会在第 8 章单独介绍。

```
case class Except(left: LogicalPlan, right: LogicalPlan) extends SetOperation(left, right) {
  override def output: Seq[Attribute] = left.output
  override protected def validConstraints: Set[Expression] = leftConstraints
  override lazy val statistics: Statistics = {
    left.statistics.copy()
  }
}
```

5.2.6 其他类型的 LogicalPlan

除上述 3 种类型的 LogicalPlan 外，在 Spark SQL 中还有 3 种直接继承自 LogicalPlan 的逻辑算子节点，如图 5.7 所示。

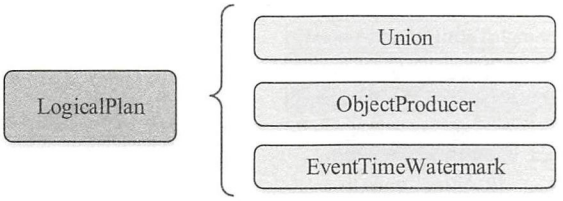


图 5.7 其他类型的 LogicalPlan

这 3 种类型的 LogicalPlan 分别是 ObjectProducer、Union 和 EventTimeWatermark 逻辑算子。其中，EventTimeWatermark 主要针对 Spark Streaming 中的 watermark 机制，一般在 SQL 中用得不多；ObjectProducer 为特质（Trait），与前面的 ObjectConsumer 相对应，用于产生只包含 Object 列的行数据；Union 算子的使用场景比较多，其子节点数目不限，是一系列 LogicalPlan 的列表。

总体来讲，LogicalPlan 中主要记录了该逻辑节点处理逻辑的相关属性，包括输入输出、约束条件、算子逻辑和统计信息等。本节对 LogicalPlan 的直接父类 QueryPlan 和各种子类进行了概述，旨在从宏观层面给出整体横向的视角，后续章节中会用特定的算子在纵向进行深入的分析。

5.3 AstBuilder 机制：Unresolved LogicalPlan 生成

介绍完不同的 LogicalPlan，现在回到本章概述中提到的转换过程，首先需要解决的是 Unresolved LogicalPlan 的生成。在第 4 章最后一节的内容中，对常用 SQL 语句生成的语法树进

行了大致描述。如何从这些 SQL 抽象语法树生成相应的逻辑算子树（Unresolved LogicalPlan）是一个关键问题，也是后续进一步转换的基础。

仍然以第 3 章案例中的 SQL 语句为例，其解析得到的抽象语法树如图 5.8 所示。Spark SQL 首先会在 ParserDriver 中通过调用语法分析器中的 singleStatement() 方法构建整棵语法树，然后通过 AstBuilder 访问者类对语法树进行访问。根据 AstBuilder 中的逻辑，其访问入口即是 visitSingleStatement 方法，该方法也是访问整棵抽象语法树的启动接口，其逻辑代码片段如下。

```
override def visitSingleStatement(
    ctx: SingleStatementContext): LogicalPlan = withOrigin(ctx) {
      visit(ctx.statement).asInstanceOf[LogicalPlan]
}
```

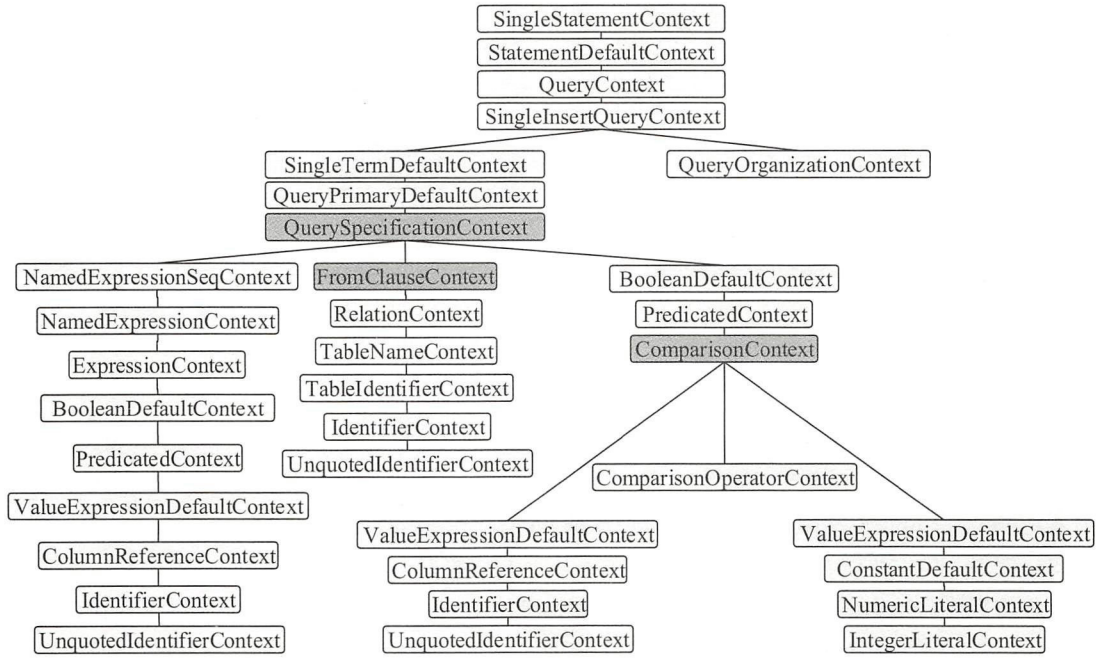


图 5.8　抽象语法树

从逻辑上来看，对根节点的访问操作会递归访问其子节点（ctx.statement，默认为 StatementDefaultContext 节点，即图 5.8 中的根节点的子节点）。这样逐步向下递归调用，直到访问某个子节点时能够构造 LogicalPlan，然后传递给父节点，因此返回的结果可以转换为 LogicalPlan 类型（注：AstBuilder 类继承基础 Visitor 类是基于 AnyRef 类型的，即 SqlBaseBaseVisitor[AnyRef]，原因是 visit 操作既可能返回 Expression 类型，也可能返回 LogicalPlan 类型）。

当整个解析过程访问到 QuerySpecificationContext 节点时，执行逻辑可以看作两部分，如下

面的代码所示：首先访问 FromClauseContext 子树，生成名为 from 的 LogicalPlan；接下来，调用 withQuerySpecification 方法在 from 的基础上完成后续扩展。

```
override def visitQuerySpecification(
      ctx: QuerySpecificationContext): LogicalPlan = withOrigin(ctx) {
    val from = OneRowRelation.optional(ctx.fromClause) {
      visitFromClause(ctx.fromClause)
    }
    withQuerySpecification(ctx, from)
}
```

总的来看，生成 Unresolved LogicalPlan 的过程如图 5.9 所示，从访问 QuerySpecificationContext 节点开始，分为以下 3 个步骤。

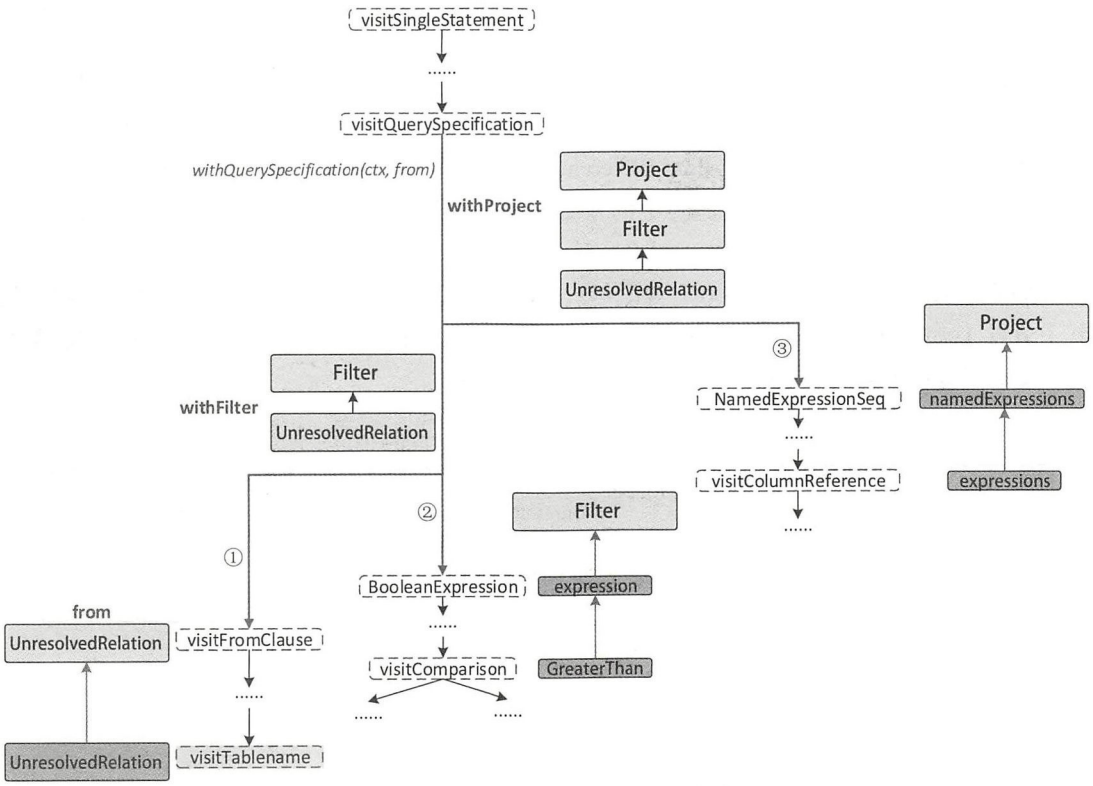


图 5.9 Unresolved LogicalPlan 的生成

（1）生成数据表对应的 LogicalPlan：访问 FromClauseContext 并递归访问，一直到匹配 TableNameContext 节点（visitTableName）时，直接根据 TableNameContext 中的数据信息生成 UnresolvedRelation，此时不再继续递归访问子节点，构造名为 from 的 LogicalPlan 并返回。

（2）生成加入了过滤逻辑的 LogicalPlan：过滤逻辑对应 SQL 中的 where 语句，在 QuerySpecificationContext 中包含了名称为 where 的 BooleanExpressionContext 类型，对应图 5.8 中的 BooleanDefaultContext 节点。AstBuilder 会对该子树进行递归访问（例如碰到 ComparisonContext 节点时会生成 GreaterThan 表达式），生成 expression 并返回作为过滤条件，然后基于此过滤条件表达式生成 Filter LogicalPlan 节点。最后，由此 LogicalPlan 和第（1）步中的 UnresolvedRelation 构造名称为 withFilter 的 LogicalPlan，其中 Filter 节点为根节点。

（3）生成加入列剪裁后的 LogicalPlan：列剪裁逻辑对应 SQL 中 select 语句对 name 列的选择操作，即图 5.9 中的最后一步操作。AstBuilder 在访问过程中会获取 QuerySpecificationContext 节点所包含的 NamedExpressionSeqContext 成员，并对其所有子节点对应的表达式进行转换，生成 NameExpression 列表（namedExpressions）然后，基于 namedExpressions 生成 Project LogicalPlan；最后，由此 LogicalPlan 和第（2）步中的 withFilter 构造名称为 withProject 的 LogicalPlan，其中 Project 最终成为整棵逻辑算子树的根节点。

为加深对此逻辑算子树的理解，有必要进一步讲解各节点中的 Expression 情况。表 5.1 列出了构造 Filter 逻辑算子树节点中的 condition 表达式（where 语句）。表中访问操作按照子节点为先的顺序，当执行 visitColumnReference 时，会根据 ColumnReferenceContext 节点信息生成 UnresolvedAttribute 表达式，其中的常数会统一封装为 Literal 表达式。在 visitPredicated 中会检查该谓词逻辑中是否包含 predicate 语句（按照文法文件中的定义，predicate 主要表示 BETWEEN-AND、IN 和 LIKE/RLIKE 等语句），这里的 SQL 不包含 predicate，因此直接返回访问其子节点（visitComparison）得到的结果。最终生成逻辑算子树 Filter 节点的 condition 构造参数为 GreaterThan 表达式，其树型结构如图 5.10（左）所示。

表 5.1　逻辑算子树 Filter 节点 condition 对应的 Expression 生成

访问操作	返回的 **Expression**
visitColumnReference	UnresolvedAttribute(Seq("AGE"))
visitIntegerLiteral	Literal（18, IntegerType）
visitComparison	GreaterThan（left, right）
visitPredicated	GreaterThan（left, right）

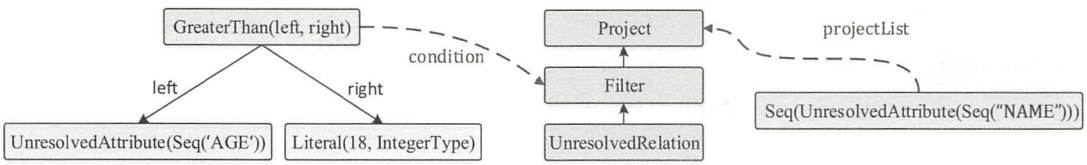


图 5.10　Expression 生成

表 5.2 列出了构造 Project 逻辑算子树节点中所选取列对应的表达式。表中访问操作按照子

节点为先的顺序，当执行 visitColumnReference 时，会对 name 列生成 UnresolvedAttribute 表达式；此时 visitPredicated 中同样不包含 predicate，因此直接返回子节点中得到的 UnresolvedAttribute；最后，执行 visitNamedExpression 访问操作，顾名思义，该操作用于对选取的列进行命名，因为语句中不涉及别名（Alias）的情况，这里也是直接返回子节点生成的表达式。此外，在 SQL 语句 select 操作中因为只选取了 name 这一列，所以最终生成逻辑算子树 Project 节点的构造参数即为 Seq(UnresolvedAttribute(Seq("NAME"))) 表达式，如图 5.10（右）所示。

表 5.2 逻辑算子树 Filter 节点的 Expression 生成

访问操作	返回的 **Expression**
visitColumnReference	UnresolvedAttribute(Seq("NAME"))
visitPredicated	UnresolvedAttribute(Seq("NAME"))
visitNamedExpression	UnresolvedAttribute(Seq("NAME"))

总的来看，最终生成的 Unresolved LogicalPlan 完整地涵盖了 SQL 语句中的信息。首先是 UnresolvedRelation 叶子节点，对应未绑定元数据信息的 student 数据表。从实现代码可以看到，其 resolved 属性设定为 false，重载实现 QueryPlan 的 output 输出设定为空，案例中的 tableName 方法返回 student 字符串。

```
case class UnresolvedRelation(tableIdentifier: TableIdentifier,
   alias: Option[String] = None) extends LeafNode {
  def tableName: String = tableIdentifier.unquotedString
  override def output: Seq[Attribute] = Nil
  override lazy val resolved = false
}
```

过滤节点 Filter 继承自 UnaryNode，从下面的代码实现可以看到，主要的方法都是直接调用子节点中的方法。注意 validateConstraints 中的实现，会将 condition 表达式中的谓词逻辑与子节点中的约束整合。

```
case class Filter(condition: Expression, child: LogicalPlan)
  extends UnaryNode with PredicateHelper {
  override def output: Seq[Attribute] = child.output
  override def maxRows: Option[Long] = child.maxRows
  override protected def validConstraints: Set[Expression] = {
    val predicates = splitConjunctivePredicates(condition)
      .filterNot(SubqueryExpression.hasCorrelatedSubquery)
    child.constraints.union(predicates.toSet)
  }
}
```

最后是列剪裁节点 Project。Project 同样继承自 UnaryNode。根据如下代码的具体实现，projectList 代表要选取列的列表，其中列表中每个表达式的类型都是 NamedExpression 类型。不

同于 Filter 节点的是，其 output 方法会直接输出 projectList 中的列，不需要考虑子节点的相关信息。从 resolved 的逻辑来看，判断一个 Project 节点是否 resolved 的条件既要满足所有表达式都已经解析，又要确认所有子节点已经解析且不包含特殊的表达式。类似的，validateConstraints 中的约束条件，会将 projectList 对应的别名约束与子节点中的约束整合。

```
case class Project(projectList: Seq[NamedExpression], child: LogicalPlan) extends UnaryNode {
  override def output: Seq[Attribute] = projectList.map(_.toAttribute)
  override def maxRows: Option[Long] = child.maxRows
  override lazy val resolved: Boolean = {
    val hasSpecialExpressions = projectList.exists ( _.collect {
        case agg: AggregateExpression => agg
        case generator: Generator => generator
        case window: WindowExpression => window
      }.nonEmpty
    )
    !expressions.exists(!_.resolved) && childrenResolved && !hasSpecialExpressions
  }
  override def validConstraints: Set[Expression] =
      child.constraints.union(getAliasedConstraints(projectList))
}
```

至此，从实例 SQL 所对应的抽象语法树生成 Unresolved LogicalPlan 的整个过程结束。基于原理分析的目的，所选取的案例比较简单，因此生成 Unresolved LogicalPlan 过程的步骤不多，上述逻辑的具体实现可以参考 AstBuilder 类中的 withQuerySpecification 方法。在实际应用中，查询往往包含更多的操作算子，所生成的逻辑算子树结构也会非常复杂，展开描述会显得过于烦琐。对于开发人员来说，重点在于熟悉 SqlBase.g4 文法文件，并抓住 AstBuilder 访问过程中的主线。

5.4 Analyzer 机制：Analyzed LogicalPlan 生成

经过上一个阶段 AstBuilder 的处理，已经得到了 Unresolved LogicalPlan。从图 5.10 中可以看到，该逻辑算子树中未被解析的有 UnresolvedRelation 和 UnresolvedAttribute 两种对象。实际上，Analyzer 所起到的主要作用就是将这两种节点或表达式解析成有类型的（Typed）对象。在此过程中，需要用到 Catalog 的相关信息。这也可以从 Analyzer 的构造参数看出。在分析 Analyzer 之前，本节先对 Spark SQL 中随处都会用到的 Catalog 体系和 Rule 体系进行介绍。

5.4.1 Catalog 体系分析

按照 SQL 标准的解释，在 SQL 环境下 Catalog 和 Schema 都属于抽象概念。在关系数据库中，Catalog 是一个宽泛的概念，通常可以理解为一个容器或数据库对象命名空间中的一个层

次，主要用来解决命名冲突等问题。

在 Spark SQL 系统中，Catalog 主要用于各种函数资源信息和元数据信息（数据库、数据表、数据视图、数据分区与函数等）的统一管理。Spark SQL 的 Catalog 体系涉及多个方面，不同层次所对应的关系如图 5.11 所示。

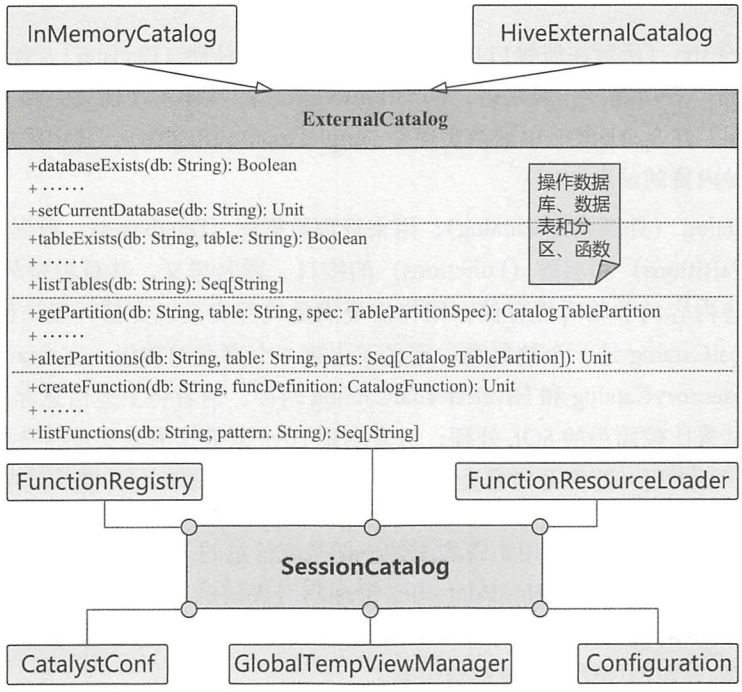


图 5.11 Catalog 体系

具体来讲，Spark SQL 中的 Catalog 体系实现以 SessionCatalog 为主体，通过 SparkSession（Spark 程序入口）提供给外部调用。一般一个 SparkSession 对应一个 SessionCatalog。本质上，SessionCatalog 起到了一个代理的作用，对底层的元数据信息、临时表信息、视图信息和函数信息进行了封装。如图 5.11 所示，SessionCatalog 的构造参数包括 6 部分，除传入 Spark SQL 和 Hadoop 配置信息的 CatalystConf 与 Configuration 外，还涉及以下 4 个方面的内容。

- GlobalTempViewManager（全局的临时视图管理）：对应 DataFrame 中常用的 createGlobalTempView 方法，进行跨 Session 的视图管理。GlobalTempViewManager 是一个线程安全的类，提供了对全局视图的原子操作，包括创建、更新、删除和重命名等。在 GlobalTempViewManager 内部实现中，主要功能依赖一个 mutable 类型的 HashMap 来对视图名和数据源进行映射，其中的 key 是视图名的字符串，value 是视图所对应的 LogicalPlan（一般在创建该视图时生成）。需要注意的是，GlobalTempViewManager 对视图名是大小写敏感的。

- FunctionResourceLoader（函数资源加载器）：在 Spark SQL 中除内置实现的各种函数外，还支持用户自定义的函数和 Hive 中的各种函数。这些函数往往通过 Jar 包或文件类型提供，FunctionResourceLoader 主要就是用来加载这两种类型的资源以提供函数的调用。需要注意的是，对于 Archive 类型的资源，目前仅支持在 YARN 模式下以 spark-submit 方式提交时进行加载。
- FunctionRegistry（函数注册接口）：用来实现对函数的注册（Register）、查找（Lookup）和删除（Drop）等功能。一般来讲，FunctionRegistry 的具体实现需要是线程安全的，以支持并发访问。在 Spark SQL 中默认实现是 SimpleFunctionRegistry，其中采用 Map 数据结构注册了各种内置的函数。
- ExternalCatalog（外部系统 Catalog）：用来管理数据库（Databases）、数据表（Tables）、数据分区（Partitions）和函数（Functions）的接口。顾名思义，其目标是与外部系统交互，并做到上述内容的非临时性存储，同样需要满足线程安全以支持并发访问。如图 5.11 所示，ExternalCatalog 是一个抽象类，定义了上述 4 个方面的功能。在 Spark SQL 中，具体实现有 InMemoryCatalog 和 HiveExternalCatalog 两种。前者将上述信息存储在内存中，一般用于测试或比较简单的 SQL 处理；后者利用 Hive 原数据库来实现持久化的管理，在生产环境中广泛应用（这部分内容会在第 10 章进行详细介绍）。

总体来看，SessionCatalog 是用于管理上述一切基本信息的入口。除上述的构造参数外，其内部还包括一个 mutable 类型的 HashMap 用来管理临时表信息，以及 currentDb 成员变量用来指代当前操作所对应的数据库名称。SessionCatalog 在 Spark SQL 的整个流程中起着重要的作用，在后续逻辑算子阶段和物理算子阶段都会用到。

5.4.2 Rule 体系

在 Unresolved LogicalPlan 逻辑算子树的操作（如绑定、解析、优化等）中，主要方法都是基于规则（Rule）的，通过 Scala 语言模式匹配机制（Pattern-match）进行树结构的转换或节点改写。Rule 是一个抽象类，子类需要复写 apply(plan: TreeType) 方法来制定特定的处理逻辑，基本定义如下。

```
abstract class Rule[TreeType <: TreeNode[_]] extends Logging {
  val ruleName: String = {
    val className = getClass.getName
    if (className endsWith "$") className.dropRight(1) else className
  }
  def apply(plan: TreeType): TreeType
}
```

不同的 Rule 实现将在 5.4.3小节和 5.5.2小节介绍。有了各种具体规则后，还需要驱动程序来调用这些规则，在 Catalyst 中这个功能由 RuleExecutor 提供。凡是涉及树型结构的转换过程（如 Analyzer 逻辑算子树分析过程、Optimizer 逻辑算子树的优化过程和后续物理算子树的生成过程等），都要实施规则匹配和节点处理，都继承自 RuleExecutor[TreeType] 抽象类，如图 5.12 所示。

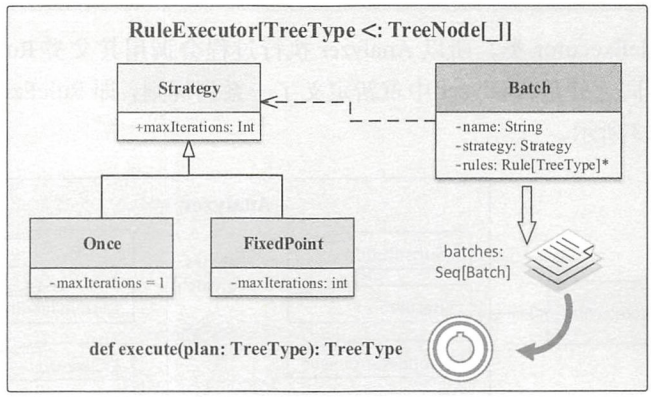


图 5.12　RuleExecutor 规则驱动

RuleExecutor 内部提供了一个 Seq[Batch]，里面定义的是该 RuleExecutor 的处理步骤。每个 Batch 代表一套规则，配备一个策略，该策略说明了迭代次数（一次还是多次）。RuleExecutor 的 apply(plan: TreeType): TreeType 方法会按照 batches 顺序和 batch 内的 Rules 顺序，对传入的 plan 里的节点进行迭代处理，处理逻辑由具体 Rule 子类实现。

```
def execute(plan: TreeType): TreeType = {
    var curPlan = plan
    batches.foreach { batch =>
      val batchStartPlan = curPlan
      var iteration = 1
      var lastPlan = curPlan
      var continue = true
      while (continue) {
        curPlan = batch.rules.foldLeft(curPlan) {
          case (plan, rule) => rule(plan)
        }
        iteration += 1
        if (iteration > batch.strategy.maxIterations) {
          continue = false
        }
        if (curPlan.fastEquals(lastPlan)) {
          continue = false
        }
        lastPlan = curPlan
      }
```

```
    }
    curPlan
}
```

5.4.3 Analyzed LogicalPlan 生成过程

因为继承自 RuleExecutor 类，所以 Analyzer 执行过程会调用其父类 RuleExecutor 中实现的 run 方法，主要的不同之处是 Analyzer 中重新定义了一系列规则，即 RuleExecutor 类中的成员变量 batches，如图 5.13 所示。

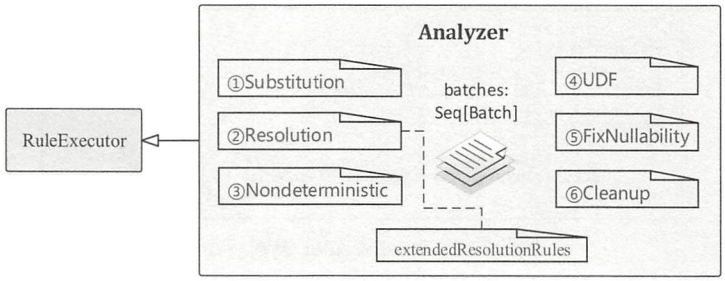


图 5.13　Analyzer 中的规则

在 Spark 2.1 版本中，Analyzer 默认定义了 6 个 Batch，共有 34 条内置的规则外加额外实现的扩展规则（图 5.13 中 extendedResolutionRules）。在分析 Analyzed LogicalPlan 生成过程之前，先对这些 Batch 进行简单的介绍，读者可结合代码阅读。

Note：Analyzer 中用到的规则比较多，因篇幅所限不方便一一展开分析。本小节对这些规则仅做概述性的分析，从宏观层面介绍规则所起到的主要作用，旨在把握规则体系的轮廓，后续章节在具体的查询分析时会对其中常用的重要规则进行讲解。

（1）Batch Substitution

顾名思义，Substitution 含义是替换，因此这个 Batch 对节点的作用类似于替换操作。目前在 Substitution 这个 Batch 中，定义了 4 条规则，分别是 CTESubstitution、WindowsSubstitution、EliminateUnions 和 SubstituteUnresolvedOrdinals。

- CTESubstitution：CTE 对应的是 With 语句，在 SQL 中主要用于子查询模块化，因此 CTESubstitution 规则也就是用来处理 With 语句的。在遍历逻辑算子树的过程中，当匹配到 With(child, relations) 节点时，将子 LogicalPlan 替换成解析后的 CTE。由于 CTE 的存在，SparkSqlParser 对 SQL 语句从左向右解析后会产生多个 LogicalPlan。这条规则的作用是将多个 LogicalPlan 合并成一个 LogicalPlan。

- WindowsSubstitution：对当前的逻辑算子树进行查找，当匹配到 WithWindowDefinition (windowDefinitions, child) 表达式时，将其子节点中未解析的窗口函数表达式（UnresolvedWindowExpression）转换成窗口函数表达式（WindowExpression）。
- EliminateUnions：在 Union 算子节点只有一个子节点时，Union 操作实际上并没有起到作用，这种情况下需要消除该 Union 节点。该规则在遍历逻辑算子树过程中，匹配到 Union(children) 且 children 的数目只有 1 个时，将 Union(children) 替换为 children.head 节点。
- SubstituteUnresolvedOrdinals：Spark 从 2.0 版本开始，在“Order By”和“Group By”语句中开始支持用常数来表示列的下标。例如，假设某行数据包括 A、B、C 3 列，那么 1 对应 A 列，2 对应 B 列，3 对应 C 列；此时“Group By 1, 2”等价于“Group By A, B”语句。而在 2.0 版本之前，这种写法会直接被当作常数而忽略。新版本中这种特性通过配置参数“spark.sql.orderByOrdinal”和“spark.sql.groupByOrdinal”进行设置，默认都为 true，表示该特性开启。SubstituteUnresolvedOrdinals 这条规则的作用就是根据这两个配置参数将下标替换成 UnresolvedOrdinal 表达式，以映射到对应的列。

（2）Batch Resolution

该 Batch 中包含了 Analyzer 中最多同时也最常用的解析规则，如表 5.3 所示。表中规则从上到下的顺序也是规则被 RuleExecutor 执行的顺序。

根据表 5.3 可知，Resolution 中加入了 25 条分析规则，以及一个 extendedResolutionRules 扩展规则列表用来支持 Analyzer 子类在扩展规则列表中添加新的分析规则。整体上来讲，表 5.3 中的这些规则涉及了常见的数据源、数据类型、数据转换和处理操作等。根据规则名称很容易看出，这些规则都针对特定的算子节点，例如 ResolveUpCast 规则用于 DataType 向 DataType 的数据类型转换。考虑到后续具体查询分析中会涉及这些规则，因此这里不展开分析。

（3）Batch Nondeterministic ⇒ PullOutNondeterministic

该 Batch 中仅包含 PullOutNondeterministic 这一条规则，主要用来将 LogicalPlan 中非 Project 或非 Filter 算子的 nondeterministic（不确定的）表达式提取出来，然后将这些表达式放在内层的 Project 算子中或最终的 Project 算子中。

（4）Batch UDF ⇒ HandleNullInputsForUDF

对于 UDF 这个规则，Batch 主要用来对用户自定义函数进行一些特别的处理，该 Batch 在 Spark 2.1 版本中仅有 HandleNullInputsForUDF 这一条规则。HandleNullInputsForUDF 规则用来处理输入数据为 Null 的情形，其主要思想是从上至下进行表达式的遍历 (transformExpressionsUp)，当匹配到 ScalaUDF 类型的表达式时，会创建 If 表达式来进行 Null 值的检查。

表 5.3　Batch Resolution 中的规则

解析规则	转换操作
ResolveTableValuedFunctions	解析可以作为数据表的函数（例如 range）
ResolveRelations	解析数据表
ResolveReferences	解析列
ResolveCreateNamedStruct	解析结构体创建
ResolveDeserializer	解析反序列化操作类
ResolveNewInstance	解析新的实例
ResolveUpCast	解析类型转换
ResolveGroupingAnalytics	解析多维分析
ResolvePivot	解析 Pivot
ResolveOrdinalInOrderByAndGroupBy	解析下标聚合
ResolveMissingReferences	解析新的列
ExtractGenerator	解析生成器
ResolveGenerate	解析生成过程
ResolveFunctions	解析函数
ResolveAliases	解析别名
ResolveSubquery	解析子查询
ResolveWindowOrder	解析窗口函数排序
ResolveWindowFrame	解析窗口函数
ResolveNaturalAndUsingJoin	解析自然 Join
ExtractWindowExpressions	提取窗口函数表达式
GlobalAggregates	解析全局聚合
ResolveAggregateFunctions	解析聚合函数
TimeWindowing	解析时间窗口
ResolveInlineTables	解析内联表
TypeCoercion.typeCoercionRules	解析强制类型转换
extendedResolutionRules	扩展规则

（5）Batch FixNullability ⇒ FixNullability

该 Batch 中仅包含 FixNullability 这一条规则，用来统一设定 LogicalPlan 中表达式的 nullable 属性。在 DataFrame 或 Dataset 等编程接口中，用户代码对于某些列（AttribtueReference）可能会改变其 nullability 属性，导致后续的判断逻辑（如 isNull 过滤等）中出现异常结果。在 FixNullability 规则中，对解析后的 LogicalPlan 执行 transformExpressions 操作，如果某列来自于其子节点，则其 nullability 值根据子节点对应的输出信息进行设置。

（6）Batch Cleanup ⇒ CleanupAliases

该 Batch 中仅包含 CleanupAliases 这一条规则，用来删除 LogicalPlan 中无用的别名信息。一般情况下，逻辑算子树中仅 Project、Aggregate 或 Window 算子的最高一层表达式（分别对应 project list、aggregate expressions 和 window expressions）才需要别名。CleanupAliases 通过 trimAliases 方法对表达式执行中的别名进行删除。

以上内容介绍的是 Spark 2.1 版本 Analyzer 中内置的分析规则整体情况，在不同版本的演

化中，这些规则也会有所变化，读者可自行分析。现在回到之前案例查询中生成的 Unresolved LogicalPlan 中。接下来的内容将会重点探讨 Analyzer 对该逻辑算子树进行分析的详细流程。

在 QueryExecution 类中可以看到，触发 Analyzer 执行的是 execute 方法，即 RuleExecutor 中的 execute 方法，该方法会循环地调用规则对逻辑算子树进行分析。

```
val analyzed: LogicalPlan = analyzer.execute(logical)
```

对于图 5.9 中的 Unresolved LogicalPlan，Analyzer 中首先匹配的是 ResolveRelations 规则。执行过程如图 5.14 所示，这也是 Analyzed LogicalPlan 生成的第 1 步。

```
object ResolveRelations extends Rule[LogicalPlan] {
    private def lookupTableFromCatalog(u: UnresolvedRelation): LogicalPlan = {
      try {
        catalog.lookupRelation(u.tableIdentifier, u.alias)
      } catch {
        case _: NoSuchTableException => u.failAnalysis(s “Table or view not found: ${u.tableName}
     ” )
      }
    }
    def apply(plan: LogicalPlan): LogicalPlan = plan resolveOperators {
      case i @ InsertIntoTable(u: UnresolvedRelation, parts, child, _, _) if child.resolved =>
        i.copy(table = EliminateSubqueryAliases(lookupTableFromCatalog(u)))
      case u: UnresolvedRelation =>
        val table = u.tableIdentifier
        if (table.database.isDefined && conf.runSQLonFile && !catalog.isTemporaryTable(table) &&
            (!catalog.databaseExists(table.database.get) || !catalog.tableExists(table))) {
          u
        } else {
          lookupTableFromCatalog(u)
        }
    }
}
```

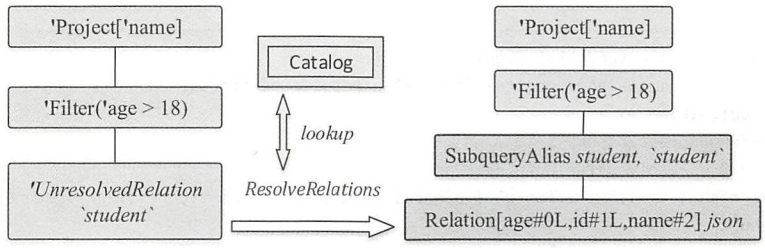


图 5.14　Analyzed LogicalPlan 生成的第 1 步

从上述 ResolveRelations 的实现中可以看到，当遍历逻辑算子树的过程中匹配到 UnresolvedRelation 节点时，对于本例会直接调用 lookupTableFromCatalog 方法从 SessionCatalog 中

查表。实际上，该表在案例 SQL 查询的上一步中就已经创建好并以 LogicalPlan 类型存储在 InMemoryCatalog 中，因此 lookupTableFromCatalog 方法直接根据其表名即可得到分析后的 LogicalPlan。

需要注意的是，在 Catalog 查表后，Relation 节点上会插入一个别名节点。此外，Relation 中列后面的数字表示下标，注意其数据类型，age 和 id 都默认设定为 Long 类型（“L”字符）。

接下来，进入第 2 步，执行 ResolveReferences 规则，得到的逻辑算子树如图 5.15 所示。可以看到，其他节点都不发生变化，主要是 Filter 节点中的 age 信息从 Unresolved 状态变成了 Analyzed 状态（表示 Unresolved 状态的前缀字符单引号已经被去掉）。

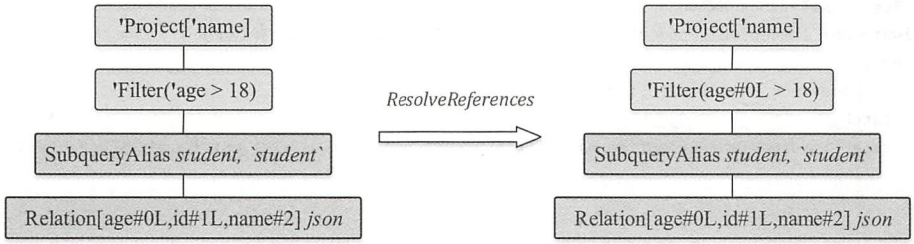


图 5.15　Analyzed LogicalPlan 生成的第 2 步

在 ResolveReferences 规则中与本例相关的匹配逻辑如以下代码所示。当碰到 UnresolvedAttribute 时，会调用 LogicalPlan 中定义的 resolveChildren 方法对该表达式进行分析。需要注意的是，resolveChildren 并不能确保一次分析成功，在分析对应表达式时，需要根据该表达式所处 LogicalPlan 节点的子节点输出信息进行判断。在对 Filter 表达式中的 age 属性进行分析时，因为 Filter 的子节点 Relation 已经处于 resolved 状态，因此可以成功；而在对 Project 中的表达式 name 属性进行分析时，因为 Project 的子节点 Filter 此时仍然处于 unresolved 状态（注：虽然 age 列完成了分析，但是整个 Filter 节点中还有“18”这个 Literal 常数表达式未被分析），因此解析操作无法成功，留待下一轮规则调用时再进行解析。

```
object ResolveReferences extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan resolveOperators {
      case q: LogicalPlan =>
        q transformExpressionsUp  {
          case u @ UnresolvedAttribute(nameParts) =>
            val result = withPosition(u) { q.resolveChildren(nameParts, resolver).getOrElse(u) }
            result
          case UnresolvedExtractValue(child, fieldExpr) if child.resolved =>
            ExtractValue(child, fieldExpr, resolver)
        }
  }
```

完成第 2 步之后会调用 TypeCoercion 规则集中的 ImplicitTypeCasts 规则，对表达式中的数据类型进行隐式转换，这是 Analyzed LogicalPlan 生成的第 3 步，如图 5.16 所示。因为在 Relation 中，age 列的数据类型为 Long，而 Filter 中的数值“18”在 Unresolved LogicalPlan 中生成的类型为 IntegerType，所以需要将“18”这个常数转换为 Long 类型。

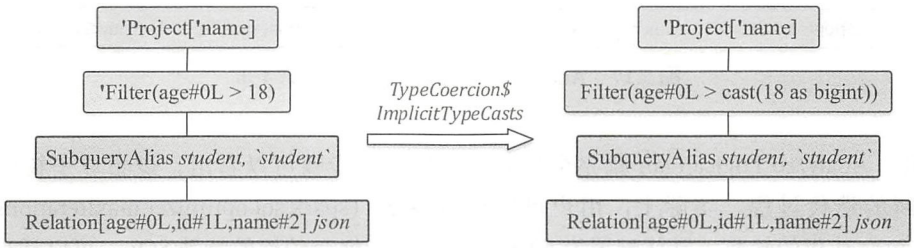


图 5.16 Analyzed LogicalPlan 生成的第 3 步

上述分析转换过程如图 5.16 所示，可以看到常数表达式“18”换为“cast(18 as bigint)”表达式（注：在 Spark SQL 类型系统中，BigInt 对应 Java 中的 Long 类型）。ImplicitTypeCasts 规则对于案例的逻辑算子树的处理过程如以下代码所示。对于 BinaryOperator 表达式，该规则会调用 findTightestCommonTypeOfTwo 找到对于左右表达式节点来讲最佳的共同数据类型。经过该规则的解析操作，可以看到图 5.16 中 Filter 节点已经变为 Analyzed 状态，节点字符前缀单引号已经被去掉。

```
object ImplicitTypeCasts extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan resolveExpressions {
      case b @ BinaryOperator(left, right) if left.dataType != right.dataType =>
        findTightestCommonTypeOfTwo(left.dataType, right.dataType).map { commonType =>
          if (b.inputType.acceptsType(commonType)) {
            val newLeft = if (left.dataType == commonType) left else Cast(left, commonType)
            val newRight = if (right.dataType == commonType) right else Cast(right, commonType)
            b.withNewChildren(Seq(newLeft, newRight))
          } else {
            b
          }
        }.getOrElse(b)
}
```

经过上述 3 个规则的解析之后，剩下的规则对逻辑算子树不起作用。此时逻辑算子树中仍然存在 Project 节点未被解析，接下来会进行下一轮规则的应用。第 4 步也是最后一步，再次执行 ResolveReferences 规则。

如图 5.17 所示，经过上一步 Filter 节点已经处于 resolved 状态，因此逻辑算子树中的 Project 节点能够完成解析。Project 节点的“name”被解析为“name#2”，其中“2”表示 name 在所有列中的下标。

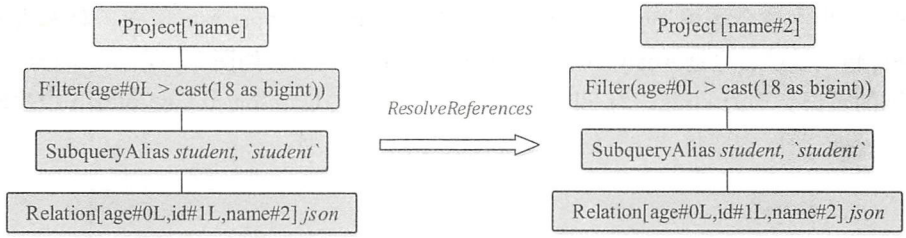


图 5.17　Analyzed LogicalPlan 生成的第 4 步

至此，Analyzed LogicalPlan 就完全生成了。从上述步骤可以看出，逻辑算子树的解析是一个不断的迭代过程。实际上，用户可以通过参数（spark.sql.optimizer.maxIterations）设定 RuleExecutor 迭代的轮数，默认配置为 50 轮，对于某些嵌套较深的特殊 SQL，可以适当地增加轮数。

5.5 Spark SQL 优化器 Optimizer

经过上一个阶段 Analyzer 的处理，Unresolved LogicalPlan 已经解析成为 Analyzed LogicalPlan。从图 5.17 中可以看出，Analyzed LogicalPlan 中自底向上节点分别对应 Relation、Subquery、Filter 和 Project 算子。

Analyzed LogicalPlan 基本上是根据 Unresolved LogicalPlan 一对一转换过来的，对于 SQL 语句中的逻辑能够很好地表示。然而，在实际应用中，很多低效的写法会带来执行效率的问题，需要进一步对 Analyzed LogicalPlan 进行处理，得到更优的逻辑算子树。于是，针对 SQL 逻辑算子树的优化器 Optimizer 应运而生。

5.5.1 Optimizer 概述

在分析 Rule 体系时就已经提到，Optimizer 同样继承自 RuleExecutor 类，本身没有重载 RuleExecutor 中的 execute 方法，因此其执行过程仍然是调用其父类 RuleExecutor 中实现的 execute 方法。在 QueryExecution 中，Optimizer 会对传入的 Analyzed LogicalPlan 执行 execute 方法，启动优化过程。

```
val optimizedPlan: LogicalPlan = optimizer.execute(analyzed)
```

与 Analyzer 类似，Optimizer 的主要机制也依赖重新定义的一系列规则，同样对应 RuleExecutor 类中的成员变量 batches，因此在 RuleExecutor 执行 execute 方法时会直接利用这些规则 Batch。

如图 5.18 所示，Optimizer 继承自 RuleExecutor，而 SparkOptimizer 又继承自 Optimizer。在上述代码中，optimizer 即是构造的 SparkOptimizer 类。从图 5.18 中可以看出，Optimizer 本身定义了 12 个规则 Batch，在 SparkOptimizer 类中又添加了 4 个 Batch。

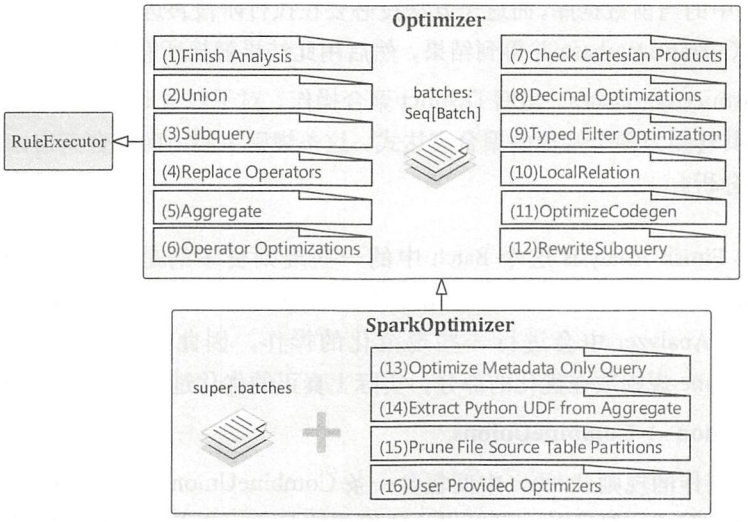


图 5.18 Optimizer 规则

5.5.2 Optimizer 规则体系

根据 5.5.1 小节的总结，Spark 2.1 版本的 SparkOptimizer 中共实现了 16 个 Batch，其中包含了 53 条优化规则，本节对这些优化规则进行系统的分析。

（1）Batch Finish Analysis

该 Batch 包含 5 条优化规则，分别是 EliminateSubqueryAliases、ReplaceExpressions、ComputeCurrentTime、GetCurrentDatabase 和 RewriteDistinctAggregates，这些规则都只执行一次。

- EliminateSubqueryAliases：消除子查询别名，对应逻辑算子树中的 SubqueryAlias 节点。一般来讲，Subqueries 仅用于提供查询的视角范围（Scope）信息，一旦 Analyzer 阶段结束，该节点就可以被移除，该优化规则直接将 SubqueryAlias 替换为其子节点。
- ReplaceExpressions：表达式替换，在逻辑算子树中查找匹配 RuntimeReplaceable 的表达式并将其替换为能够执行的正常表达式。这条规则通常用来对其他类型的数据库提供兼容的能力，例如，可以用“coalesce”来替换支持“nvl”的表达式。
- ComputeCurrentTime：计算与当前时间相关的表达式，在同一条 SQL 语句中可能包含多个计算时间的表达式，即 CurrentDate 和 CurrentTimestamp，且该表达式出现在多个语句

中。为避免不一致，ComputeCurrentTime 对逻辑算子树中的时间函数计算一次后，将其他同样的函数替换成该计算结果。

- GetCurrentDatabase：获取当前数据库，在 SQL 语句中可能会调用 CurrentDatabase 函数来获取 Catalog 中的当前数据库，而这个方法没必要在执行阶段再进行计算。GetCurrentDatabase 规则执行 CurrentDatabase 并得到结果，然后用此结果替换所有的 CurrentDatabase 表达式。
- RewriteDistinctAggregates：重写 Distinct 聚合操作，对于包含 Distinct 算子的聚合语句，这条规则将其转换为两个常规的聚合表达式。这条规则主要面向聚合查询，在第 7 章会对其进行详细分析。

严格来讲，Finish Analysis 这个 Batch 中的一些规则更多的是为了得到正确的结果（例如 ComputeCurrentTime），并不涉及优化操作，从逻辑上更应该归于 Analyzer 的分析规则中。但是考虑到 Analyzer 中会进行一些规范化的操作，因此将 EliminateSubqueryAliases 和 ComputeCurrentTime 规则放在优化的部分，实际上真正的优化过程从下一个 Batch 开始。

（2）Batch Union ⇒ CombineUnions

针对 Union 操作的规则 Batch，中间包含一条 CombineUnions 优化规则。在逻辑算子树中，当相邻的节点都是 Union 算子时，可以将这些相邻的 Union 节点合并为一个 Union 节点。在该规则中，flattenUnion 是核心方法，用栈实现了节点的合并。需要注意的是，后续的优化操作可能会将原来不相邻的 Union 节点变得相邻，因此在后面的规则 Batch 中又加入了 CombineUnions 这条规则。

（3）Batch Subquery ⇒ OptimizeSubqueries

该 Batch 目前只包含 OptimizeSubqueries 这一条优化规则。当 SQL 语句包含子查询时，会在逻辑算子树上生成 SubqueryExpression 表达式。OptimizeSubqueries 优化规则在遇到 SubqueryExpression 表达式时，进一步递归调用 Optimizer 对该表达式的子计划并进行优化。

（4）Batch Replace Operators

该 Batch 中的优化规则主要用来执行算子的替换操作。在 SQL 语句中，某些查询算子可以直接改写为已有的算子，避免进行重复的逻辑转换。Replace Operators 中包含 ReplaceIntersectWithSemiJoin、ReplaceExceptWithAntiJoin 和 ReplaceDistinctWithAggregate 这 3 条优化规则。

- ReplaceIntersectWithSemiJoin：将 Intersect 操作算子替换为 Left-Semi Join 操作算子，从逻辑上来看，这两种算子是等价的。需要注意的是，ReplaceIntersectWithSemiJoin 优化规则仅适用于 INTERSECT DISTINCT 类型的语句，而不适用于 INTERSECT ALL 语句。此外，该优化规则执行之前必须消除重复的属性，避免生成的 Join 条件不正确。

- ReplaceExceptWithAntiJoin：将 Except 操作算子替换为 Left-Anti Join 操作算子，从逻辑上来看，这两种算子是等价的。与上一条优化规则一样，ReplaceExceptWithAntiJoin 优化规则仅适用于 EXCEPT DISTINCT 类型的语句，而不适用于 EXCEPT ALL 语句。此外，该优化规则执行之前必须消除重复的属性，避免生成的 Join 条件不正确。
- ReplaceDistinctWithAggregate：该优化规则会将 Distinct 算子转换为 Aggregate 语句。在某些 SQL 语句中，Select 直接进行 Distinct 操作，这种情况下可以将其直接转换为聚合操作。ReplaceDistinctWithAggregate 规则会将 Distinct 算子替换为对应的 Group By 语句。

从以上描述中可以看出，Replace Operators 主要针对的是集合类型的操作算子。

（5）Batch Aggregate

该 Batch 主要用来处理聚合算子中的逻辑，包括 RemoveLiteralFromGroupExpressions 和 RemoveRepetitionFromGroupExpressions 两条规则。RemoveLiteralFromGroupExpressions 优化规则用来删除 Group By 语句中的常数，这些常数对于结果无影响，但是会导致分组数目变多。此外，如果 Group By 语句中全部是常数，则会将其替换为一个简单的常数 0 表达式。RemoveRepetitionFromGroupExpressions 优化规则将重复的表达式从 Group By 语句中删除，同样对结果无影响。

（6）Batch Operator Optimizations

类似 Analyzer 中的 Operator 解析规则，该 Batch 包含了 Optimizer 中数量最多同时也是最常用的各种优化规则，共 31 条。从整体来看，这 31 条优化规则（如表 5.4 所示）可以分为 3 个模块：算子下推（Operator Push Down）、算子组合（Operator Combine）、常量折叠与长度削减（Constant Folding and Strength Reduction）。

算子下推：算子下推是数据库中常用的优化方式，表 5.4 中所列的前 8 条规则都属于算子下推的模块。顾名思义，算子下推所执行的优化操作主要是将逻辑算子树中上层的算子节点尽量下推，使其靠近叶子节点，这样能够在不同程度上减少后续处理的数据量甚至简化后续的处理逻辑。以常见的列剪裁（ColumnPruning）优化为例，假设数据表中有 A、B、C 3 列，但是查询语句中只涉及 A、B 两列，那么 ColumnPruning 将会在读取数据后剪裁出这两列。又如 LimitPushDown 优化规则，能够将 LocalLimit 算子下推到 Union All 和 Outer Join 操作算子的下方，减少这两种算子在实际计算过程中需要处理的数据量。

算子组合：算子组合类型的优化规则将逻辑算子树中能够进行组合的算子尽量整合在一起，避免多次计算，以提高性能。表 5.4 中间 6 条规则（从 CollapseRepartition 到 CombineUnions）都属于算子组合类型的优化。可以看到这些规则主要针对的是重分区（repartition）算子、投影（Project）算子、过滤（Filter）算子、Window 算子、Limit 算子和 Union 算子，其中 CombineUnions 在之前已经提到过。需要注意的是，这些规则主要针对的是算子相邻的情况。

表 5.4　Batch Operator Optimizations 中的规则

优化规则	优化操作
PushProjectionThroughUnion	列剪裁下推
ReorderJoin	Join 顺序优化
EliminateOuterJoin	OuterJoin 消除
PushPredicateThroughJoin	谓词下推到 Join 算子
PushDownPredicate	谓词下推
LimitPushDown	Limit 算子下推
ColumnPruning	列剪裁
InferFiltersFromConstraints	约束条件提取
CollapseRepartition	重分区组合
CollapseProject	投影算子组合
CollapseWindow	Window 组合
CombineFilters	过滤条件组合
CombineLimits	Limit 操作组合
CombineUnions	Union 算子组合
NullPropagation	Null 提取
FoldablePropagation	可折叠算子提取
OptimizeIn	In 操作优化
ConstantFolding	常数折叠
ReorderAssociativeOperator	重排序关联算子优化
LikeSimplification	Like 算子简化
BooleanSimplification	Boolean 算子简化
SimplifyConditionals	条件简化
RemoveDispensableExpressions	Dispensable 表达式消除
SimplifyBinaryComparison	比较算子简化
PruneFilters	过滤条件剪裁
EliminateSorts	排序算子消除
SimplifyCasts	Cast 算子简化
SimplifyCaseConversionExpressions	case 表达式简化
RewriteCorrelatedScalarSubquery	依赖子查询重写
EliminateSerialization	序列化消除
RemoveAliasOnlyProject	消除别名

常量折叠与长度削减：对于逻辑算子树中涉及某些常量的节点，可以在实际执行之前就完成静态处理。常量折叠与长度削减类型的优化规则主要针对的就是这种情况。表 5.4 中的后 17 条优化规则都属于这种类型。例如，在 ConstantFolding 规则中，对于能够 foldable（可折叠）的表达式会直接在 EmptyRow 上执行 evaluate 操作，从而构造新的 Literal 表达式；PruneFilters 优化规则会详细地分析过滤条件，对总是能够返回 true 或 false 的过滤条件进行特别的处理。

（7）Batch Check Cartesian Products ⇒ CheckCartesianProducts

该 Batch 只有 CheckCartesianProducts 这一条优化规则，用来检测逻辑算子树中是否存在笛卡儿积类型的 Join 操作。如果存在这样的操作，而 SQL 语句中没有显示地使用 cross join 表达

式，则会抛出异常。CheckCartesianProducts 规则必须在 ReorderJoin 规则执行之后才能执行，确保所有的 Join 条件收集完毕。需要注意的是，当“spark.sql.crossJoin.enabled”参数设置为 true 时，该规则会被忽略。

（8）Batch Decimal Optimizations ⇒ DecimalAggregates

该 Batch 只有 DecimalAggregates 这一条优化规则，用于处理聚合操作中与 Decimal 类型相关的问题。一般情况下，如果聚合查询中涉及浮点数的精度处理，性能就会受到很大的影响。对于固定精度的 Decimal 类型，DecimalAggregates 规则将其当作 unscaled Long 类型来执行，这样可以加速聚合操作的速度。

（9）Batch Typed Filter Optimization ⇒ CombineTypedFilters

该 Batch 仅包含 CombineTypedFilters 这一条优化规则，用来对特定情况下的过滤条件进行合并。当逻辑算子树中存在两个 TypedFilter 过滤条件且针对同类型的对象条件时，CombineTypedFilters 优化规则会将它们合并到同一个过滤函数中。

（10）Batch LocalRelation ⇒ ConvertToLocalRelation | PropagateEmptyRelation

该 Batch 主要用来优化与 LocalRelation 相关的逻辑算子树，包含 ConvertToLocalRelation 和 PropagateEmptyRelation 两条优化规则。ConvertToLocalRelation 将 LocalRelation 上的本地操作（不涉及数据交互）转换为另一个 LocalRelation，目前该规则实现较为简单，仅处理 Project 投影操作。PropagateEmptyRelation 优化规则会将包含空的 LocalRelation 进行折叠。

（11）Batch OptimizeCodegen ⇒ OptimizeCodegen

该 Batch 只有 OptimizeCodegen 这一条优化规则，用来对生成的代码进行优化。代码生成技术会在与 Tungsten 相关的章节中讲解。OptimizeCodegen 规则主要针对的是 case when 语句，当 case when 语句中的分支数目不超过配置中的最大数目时，该表达式才能执行代码生成。实际上，Spark 2.1 版本中的代码生成还存在许多不足之处，更加完善的实现可以参考 2.3 版本。

（12）Batch RewriteSubquery ⇒ RewritePredicateSubquery | CollapseProject

该 Batch 主要用来优化子查询，目前包含 RewritePredicateSubquery 和 CollapseProject 两条优化规则。RewritePredicateSubquery 将特定的子查询谓词逻辑转换为 left-semi/anti join 操作。其中，EXISTS 和 NOT EXISTS 算子分别对应 semi 和 anti 类型的 Join，过滤条件会被当作 Join 的条件；IN 和 NOT IN 也分别对应 semi 和 anti 类型的 Join，过滤条件和选择的列都会被当作 join 的条件。CollapseProject 优化规则比较简单，类似 CombineTypedFilters 优化规则，会将两个相邻的 Project 算子组合在一起并执行别名替换，整合成一个统一的表达式。

（13）Batch Optimize Metadata Only Query ⇒ OptimizeMetadataOnlyQuery

该 Batch 仅执行一次，只有 OptimizeMetadataOnlyQuery 这一条规则，用来优化执行过程中只需查找分区级别元数据的语句。需要注意的是，OptimizeMetadataOnlyQuery 优化规则适用于

扫描的所有列都是分区列且包含聚合算子的情形，而且聚合算子需要满足以下情况之一：聚合表达式是分区列；分区列的聚合函数有 DISTINCT 算子；分区列的聚合函数中是否有 DISTINCT 算子不影响结果。

（14）Batch Extract Python UDF from Aggregate ⇒ ExtractPythonUDFFromAggregate

该 Batch 仅执行一次，只有 ExtractPythonUDFFromAggregate 这一条规则，用来提取出聚合操作中的 Python UDF 函数。该规则主要针对的是采用 PySpark 提交查询的情形，将参与聚合的 Python 自定义函数提取出来，在聚合操作完成之后再执行。

（15）Batch Prune File Source Table Partitions ⇒ PruneFileSourcePartitions

该 Batch 仅执行一次，只有 PruneFileSourcePartitions 这一条规则，用来对数据文件中的分区进行剪裁操作。当数据文件中定义了分区信息且逻辑算子树中的 LogicalRelation 节点上方存在过滤算子时，PruneFileSourcePartitions 优化规则会尽可能地将过滤算子下推到存储层，这样可以避免读入无关的数据分区。

（16）Batch User Provided Optimizers ⇒ ExperimentalMethods.extraOptimizations

顾名思义，该 Batch 用于支持用户自定义的优化规则，其中 ExperimentalMethods 的 extraOptimizations 队列默认为空。可以看到，Spark SQL 在逻辑算子树的转换阶段是高度可扩展的，用户只需要继承 Rule[LogicalPlan] 虚类，实现相应的转换逻辑就可以注册到优化规则队列中应用执行。

5.5.3 Optimized LogicalPlan 的生成过程

上述内容对 SparkOptimizer 中的优化规则进行了系统概述，现在回到案例对应的 Analyzed LogicalPlan。接下来，将会重点分析 Optimzer 对该逻辑算子树进行优化处理的具体流程。

对于案例生成的 Analyzed LogicalPlan，首先执行的是 Finish Analysis 这个 Batch 中的 Eliminate- SubqueryAliases 优化规则，用来消除子查询别名的情形。

EliminateSubqueryAliases 优化规则的实现逻辑如以下代码所示，可以看到，该规则的实现非常简单，直接将 SubqueryAlias 逻辑算子树节点替换为其子节点。经过 EliminateSubqueryAliases 规则优化后的逻辑算子树如图 5.19 所示。可见 SubqueryAlias 节点被删除，Filter 节点直接作用于 Relation 节点。

```
object EliminateSubqueryAliases extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transformUp {
    case SubqueryAlias(_, child, _) => child
  }
}
```

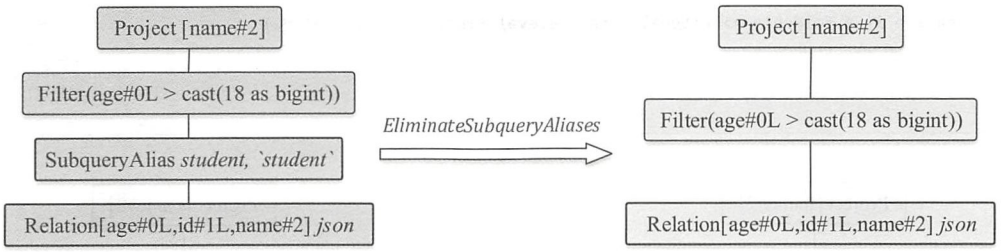


图 5.19 Optimized LogicalPlan 生成的第 1 步

第 2 步优化将匹配 Operator Optimizations 这个 Batch 中的 InferFiltersFromConstraints 优化规则，用来增加过滤条件。InferFiltersFromConstraints 优化规则会对当前节点的约束条件进行分析，生成额外的过滤条件列表，这些过滤条件不会与当前算子或其子节点现有的过滤条件重叠，具体实现如以下代码片段所示（注：案例逻辑算子树中不涉及 Join 查询语句，因此这里的代码片段中未包含 Join 算子的匹配部分）。

```
object InferFiltersFromConstraints extends Rule[LogicalPlan] with PredicateHelper {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case filter @ Filter(condition, child) =>
      val newFilters = filter.constraints --
         (child.constraints ++ splitConjunctivePredicates(condition))
      if (newFilters.nonEmpty) {
        Filter(And(newFilters.reduce(And), condition), child)
      } else {
        filter
      }
  }
}
```

从上述代码逻辑可知，对于上一步生成的逻辑算子树中的 Filter 节点，会构造新的过滤条件（newFilter）。当新的过滤条件不为空时，会与现有的过滤条件进行整合，构造新的 Filter 逻辑算子节点。

经过 InferFiltersFromConstraints 规则优化之后的逻辑算子树如图 5.20 所示，Filter 逻辑算子树节点中多了“isnotnull(age#0L)”这个过滤条件。该过滤条件来自于 Filter 中的约束信息，用来确保筛选出来的数据 age 字段不为 null。

最后一步，上述逻辑算子树会匹配 Operator Optimizations 这个 Batch 中的 ConstantFolding 优化规则，对 LogicalPlan 中可以折叠的表达式进行静态计算直接得到结果，简化表达式。

```
object ConstantFoldingFolding extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case q: LogicalPlan => q transformExpressionsDown {
      case l: Literal => l
```

```
      case e if e.foldable => Literal.create(e.eval(EmptyRow), e.dataType)
    }
  }
}
```

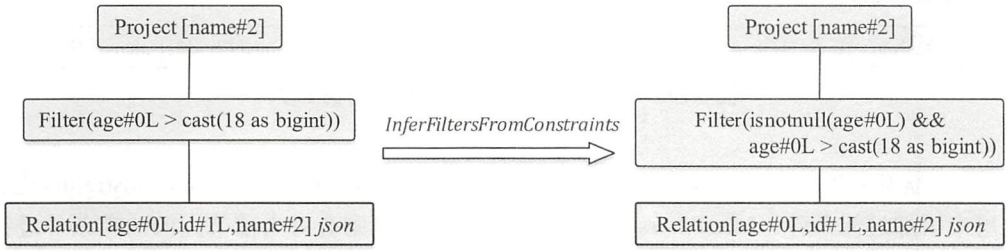


图 5.20　Optimized LogicalPlan 生成的第 2 步

在 ConstantFolding 规则中，如果 LogicalPlan 中的表达式可以折叠（foldable 为 true），那么会将 EmptyRow 作为参数传递到其 eval 方法中直接计算，然后根据计算结果构造 Literal 常量表达式。经过该规则优化后的逻辑算子树如图 5.21 所示。

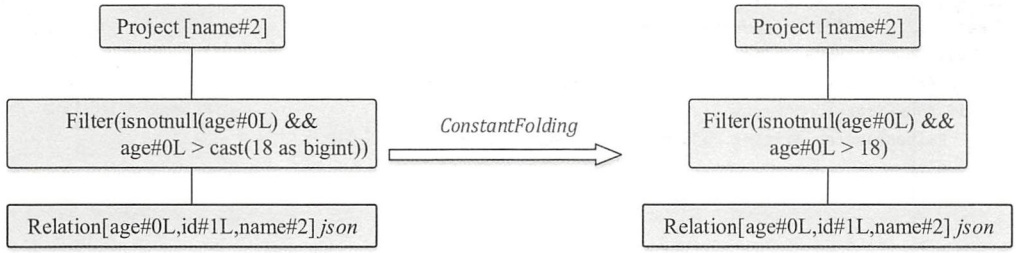


图 5.21　Optimized LogicalPlan 生成的第 3 步

可见，Filter 过滤条件中的“cast(18, bigint)”表达式经过计算成为“Literal(18, bigint)”表达式，即输出的结果为 18。在原来的 Cast 表达式中，其子节点 Literal 表达式的 foldable 值为 true，因此 Cast 表达式本身的 foldable 值也为 true，在匹配该优化规则时，Cast 表达式会被直接计算。

经过上述步骤，Spark SQL 逻辑算子树生成、分析与优化的整个阶段都执行完毕。最终生成的逻辑算子树包含 Relation 节点、Filter 节点和 Project 节点，同时每个节点中又包含了由对应表达式构成的树。这棵逻辑算子树将作为 Spark SQL 中生成物理算子树的输入，开始下一个阶段。

5.6 本章小结

在 Spark SQL 中，逻辑计划阶段起着承上启下的作用。对于用户的 SQL 语句，逻辑算子树既要完全涵盖查询与处理中的语义逻辑，又要定义各种体系的数据结构，并将这些数据结构整

合在一起。对于后续的物理算子阶段，逻辑算子树需要提供完整的逻辑信息，因此这个阶段涉及了方方面面的内容，非常庞大。

本章的目的在于从全局的视角加深读者对 LogicalPlan 阶段的理解。内容上包含了两方面，一方面是各个步骤的横向分析，包括逻辑算子树所经过的 Unresolved LogicalPlan、Analyzed LogicalPlan 和 Optimized LogicalPlan 这 3 个通用阶段，涉及 Catalog 体系、分析规则体系等；另一方面结合案例纵向地对每个过程进行了详细的阐述。即便如此，案例较为简单，涉及的算子有限，难以覆盖各种情况。对于更多的查询类型，会在后续章节进行探讨。

第6章 Spark SQL 物理计划（PhysicalPlan）

物理计划阶段是 Spark SQL 整个查询处理流程的最后一步。不同于逻辑计划（LogicalPlan）的平台无关性，物理计划（PhysicalPlan）是与底层平台紧密相关的。在此阶段，Spark SQL 会对生成的逻辑算子树进行进一步处理，得到物理算子树，并将 LogicalPlan 节点及其所包含的各种信息映射成 Spark Core 计算模型的元素，如 RDD、Transformation 和 Action 等，以支持其提交执行。

6.1 Spark SQL 物理计划概述

在 Spark SQL 中，物理计划用 SparkPlan 表示，从 Optimized LogicalPlan 传入到 Spark SQL 物理计划提交并执行，主要经过 3 个阶段。如图 6.1 所示，这 3 个阶段分别产生 Iterator[PhysicalPlan]、SparkPlan 和 Prepared SparkPlan，其中 Prepared SparkPlan 可以直接提交并执行（注：这里的“PhysicalPlan”和“SparkPlan”均表示物理计划）。

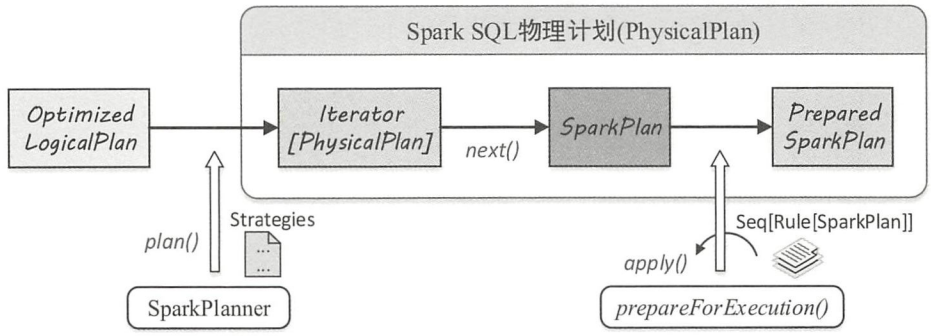


图 6.1 物理计划概述

具体来讲，这 3 个阶段所做的工作分别如下。

（1）由 SparkPlanner 将各种物理计划策略（Strategy）作用于对应的 LogicalPlan 节点上，生成 SparkPlan 列表（注：一个 LogicalPlan 可能产生多种 SparkPlan）。

（2）选取最佳的 SparkPlan，在 Spark 2.1 版本中的实现较为简单，在候选列表中直接用 next() 方法获取第一个。

（3）提交前进行准备工作，进行一些分区排序方面的处理，确保 SparkPlan 各节点能够正确执行，这一步通过 prepareForExecution() 方法调用若干规则（Rule）进行转换。

本章 6.2 节对 SparkPlan 所涉及的内容进行全面介绍，包括 SparkPlan 的分类和各种体系的概述；接下来，6.3 节与 6.4 节会重点分析 Metadata 与 Metric 体系，以及 Partitioning 与 Ordering 体系；6.5 节会详细分析 SparkPlanner 生成 SparkPlan 的过程与策略；最后，6.6 节选取若干规则阐述执行前准备的相关处理工作。

6.2 SparkPlan 简介

Spark SQL 最终将 SQL 语句经过逻辑算子树转换成物理算子树。在物理算子树中，叶子类型的 SparkPlan 节点负责“从无到有”地创建 RDD，每个非叶子类型的 SparkPlan 节点等价于在 RDD 上进行一次 Transformation，即通过调用 execute() 函数转换成新的 RDD，最终执行 collect() 操作触发计算，返回结果给用户。

如图 6.2 所示，SparkPlan 在对 RDD 做 Transformation 的过程中除对数据进行操作外，还可能对 RDD 的分区做调整。此外，SparkPlan 除实现 execute 方法外，还有一种情况是直接执行 executeBroadcast 方法，将数据广播到集群上。

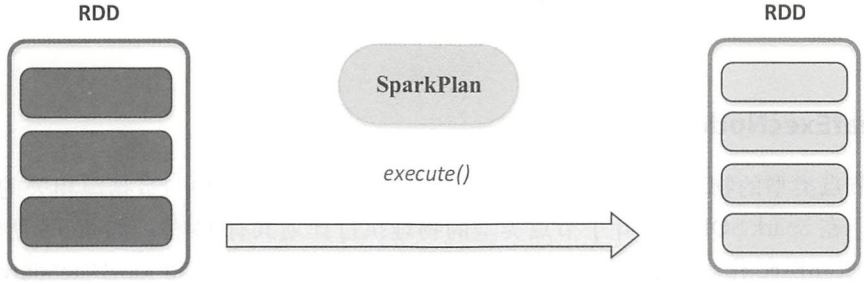


图 6.2 SparkPlan 操作

具体来看，SparkPlan 的主要功能可以划分为 3 大块。首先，每个 SparkPlan 节点必不可少地会记录其元数据（Metadata）与指标（Metric）信息，这些信息以 Key-Value 的形式保存在 Map 数据结构中，统称为 SparkPlan 的 Metadata 与 Metric 体系。其次，在对 RDD 进行 Transformation 操作时，会涉及数据分区（Partitioning）与排序（Ordering）的处理，称为 SparkPlan 的 Partitioning

与 Ordering 体系；最后，SparkPlan 作为物理计划，支持提交到 Spark Core 去执行，即 SparkPlan 的执行操作部分，以 execute 和 executeBroadcast 方法为主。此外，SparkPlan 中还定义了一些辅助函数，如创建新谓词的 newPredicate 等，这些细节本章不再专门讲解。

在 Spark 2.1 版本中，Spark SQL 大约包含 65 种具体的 SparkPlan 实现，涉及数据源 RDD 的创建和各种数据处理等。根据 SparkPlan 的子节点数目，可以大致将其分为 4 类。如图 6.3 所示，分别为 LeafExecNode、UnaryExecNode、BinaryExecNode 和其他不属于这 3 种子节点的类型，下面分别对这几种类型进行简要介绍。

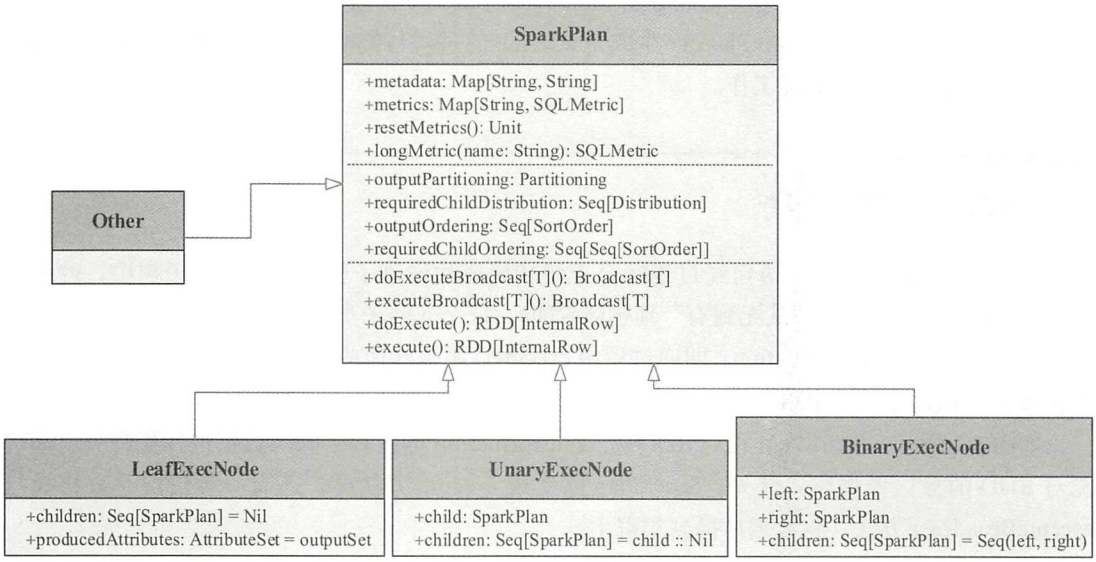


图 6.3　SparkPlan 基本类型

6.2.1 LeafExecNode 类型

叶子节点类型的物理执行计划不存在子节点。物理执行计划中与数据源相关的节点都属于该类型。在 Spark SQL 中，叶子节点类型的物理执行计划共有 13 种，如图 6.4 所示。其中，DataSourceScanExec 作为基类，具体的实现包括 FileSourceScanExec 和 RawDataSourceScanExec 两种。

LeafExecNode 类型的 SparkPlan 负责对初始 RDD 的创建。例如，RangeExec 会利用 SparkContext 中的 parallelize 方法生成给定范围内的 64 位数据的 RDD，HiveTableScanExec 会根据 Hive 数据表存储的 HDFS 信息直接生成 HadoopRDD，FileSourceScanExec 根据数据表所在的源文件生成 FileScanRDD。

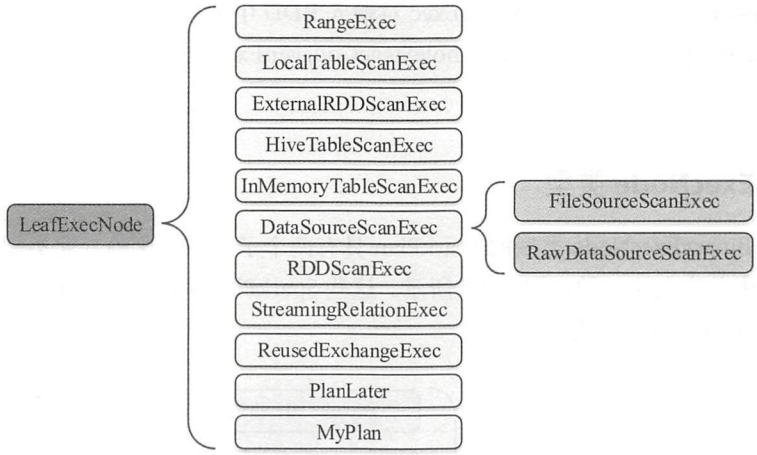


图 6.4 LeafExecNode 类型的 SparkPlan

6.2.2 UnaryExecNode 类型

UnaryExecNode 类型的物理执行计划的节点是一元的，意味着只包含 1 个子节点。在 Spark 2.1 版本中，UnaryExecNode 类型的物理执行计划共有 37 种，如图 6.5 所示。实际上，UnaryExecNode 类型的物理计划也是数量最多的类型。

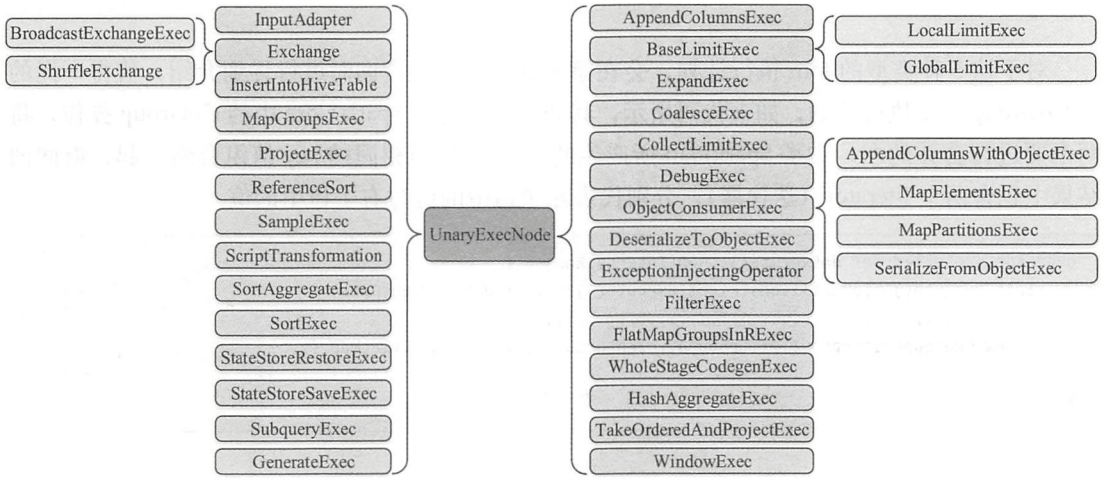


图 6.5 UnaryExecNode 类型的 SparkPlan

UnaryExecNode 节点的作用主要是对 RDD 进行转换操作。例如，第 3 章案例所生成的物理算子树中，ProjectExec 和 FilterExec 分别对子节点产生的 RDD 进行列剪裁与行过滤操作。

Exchange 负责对数据进行重分区，SampleExec 对输入 RDD 中的数据进行采样，SortExec 按照一定条件对输入 RDD 中数据进行排序，WholeStageCodegenExec 类型的 SparkPlan 将生成的代码整合成单个 Java 函数。

6.2.3 BinaryExecNode 类型

顾名思义，BinaryExecNode 类型的 SparkPlan 具有两个子节点，这种二元类型的物理执行计划在 Spark SQL 中共定义了 6 种，如图 6.6 所示。这些 SparkPlan 中除 CoGroupExec 外，其余的 5 种都是不同类型的 Join 执行计划。

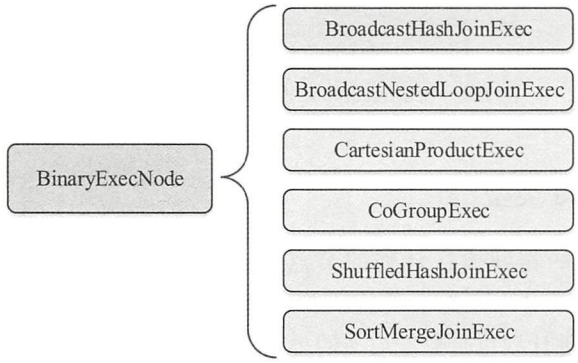


图 6.6　BinaryExecNode 类型的 SparkPlan

对于这 5 种类型的 Join 执行计划，会在第 8 章分析 Join 查询时进行详细介绍。值得一提的是 CoGroupExec 执行计划，如下代码所示，其处理逻辑类似 Spark Core 中的 CoGroup 操作，将两个要进行合并的左、右子 SparkPlan 所产生的 RDD，按照相同的 key 值组合到一起，返回的结果中包含两个 Iterator（迭代器），分别代表左子树中的值与右子树中的值。

```
override protected def doExecute(): RDD[InternalRow] = {
    left.execute().zipPartitions(right.execute()) { (leftData, rightData) =>
      ..................................................
      new CoGroupedIterator(leftGrouped, rightGrouped, leftGroup).flatMap {...}
    }
}
```

6.2.4 其他类型的 SparkPlan

除上述 3 种类型的 SparkPlan 外，Spark SQL 中还有 11 个其他类型的物理执行计划。如图 6.7 所示，这 10 种 SparkPlan 中除 CodeGenSupport 和 UnionExec 外，其他几种用到的场景并不多见。

例如，DummySparkPlan、FastOperator 和 MyPlan 均出现在单元测试中，其中 DummySparkPlan 对每个成员赋予默认值，MyPlan 则用于在 Driver 端更新 Metric 信息。

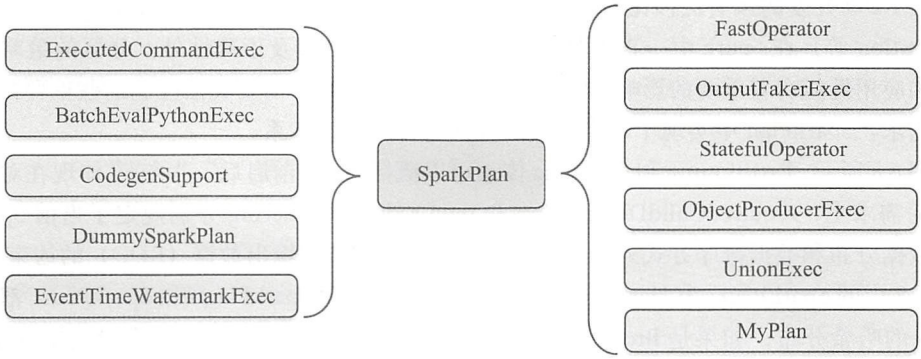


图 6.7 其他类型的 SparkPlan

6.3 Metadata 与 Metrics 体系

元数据和指标信息是性能优化的基础，SparkPlan 提供了 Map 类型的数据结构来存储相关信息，以便更加详细地刻画 SparkPlan 的细节。默认情况下，SparkPlan 中这两个 Map 的值均为空。

元数据信息 Metadata 对应 Map 中的 key 和 value 都为字符串类型。一般情况下，元数据主要用于描述数据源的一些基本信息，例如数据文件的格式、存储路径等。目前只有 FileSourceScanExec 和 RowDataSourceScanExec 两种叶子节点类型的 SparkPlan 对其进行了重载实现。

指标信息 Metrics 对应 Map 中的 key 为字符串类型，而 value 部分是 SQLMetrics 类型。在 Spark 执行过程中，Metrics 能够记录各种信息，为应用的诊断和优化提供基础。目前，Spark SQL 中共有 27 个 SparkPlan 重载实现了该方法。例如，FilterExec 中添加了“numOutputRows”指标，记录输出的数据数目，该指标会随着对应的 SparkPlan 执行而计算；ShuffleExchange 中添加了“dataSize”指标，能够记录进行重新分区操作过程中的数据总量。

在对 Spark SQL 进行定制时，用户可以自定义一些指标，并将这些指标显示在 UI 上。一方面，定义越多的指标会得到越详细的信息；另一方面，指标信息要随着执行过程而不断更新，会导致额外的计算，在一定程度上影响性能。

6.4 Partitioning 与 Ordering 体系

熟悉 RDD 开发的读者应该比较清楚，分区（Partitioning）和排序（Ordering）相关的操作（如 repartition 等）在 Spark 中一直都是较为重要的内容。除涉及正确性外，分区的策略还对集群资源和应用性能有着重要的影响。Spark SQL 作为高层模块，需要灵活控制这些重要操作。针对这种需求，SparkPlan 中实现了较为完整的分区与排序操作体系。

如图 6.8 所示，Partitioning 和 Ordering 体系可以概括为“承前启后”。“承前”体现在对输入数据特性的需求上，requiredChildDistribution 和 requiredChildOrdering 分别规定了当前 SparkPlan 所需的数据分布和数据排序方式列表，本质上是对所有子节点输出数据（RDD）的约束。例如，假设图 6.8 中的 SparkPlan2 为 Hash 类型的 Join，那么就需要 SparkPlan0 和 SparkPlan1 都是基于相同 key 的哈希分布；如果是 Broadcast 类型的 Join，那么必有一个为广播变量数据分布。“启后”体现在对输出数据的操作上，outputPartitioning 定义了当前 SparkPlan 对输出数据（RDD）的分区操作，outputOrdering 则定义了每个数据分区的排序方式。

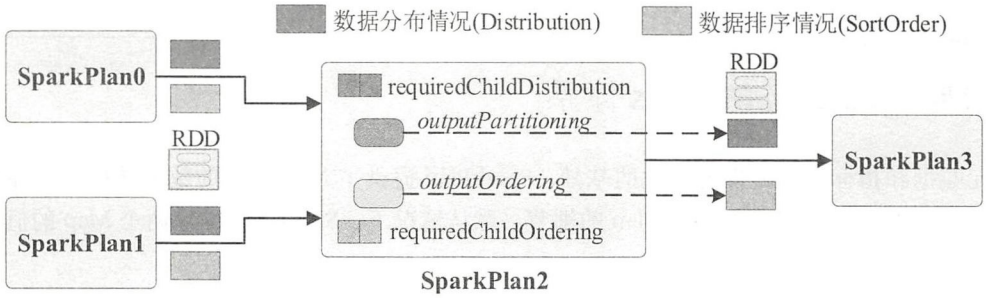


图 6.8　Partitioning 与 Ordering 体系概览

整体来看，Partitioning 与 Ordering 体系在实现上都是“前后关联”的，当前 SparkPlan 节点的操作影响着下一个 SparkPlan 节点所需的输入数据特性。考虑到 Ordering 体系仅涉及排序，实现上较为简单，因此这里不再展开分析，接下来的内容将聚焦在 Partitioning 体系的实现上。

6.4.1 Distribution 与 Partitioning 的概念

在 SparkPlan 分区体系实现中，Partitioning 表示对数据进行分区的操作，Distribution 则表示数据的分布。在 Spark SQL 中 Distribution 与 Partitioning 均被定义为接口，其具体实现有多个类，如图 6.9 所示。

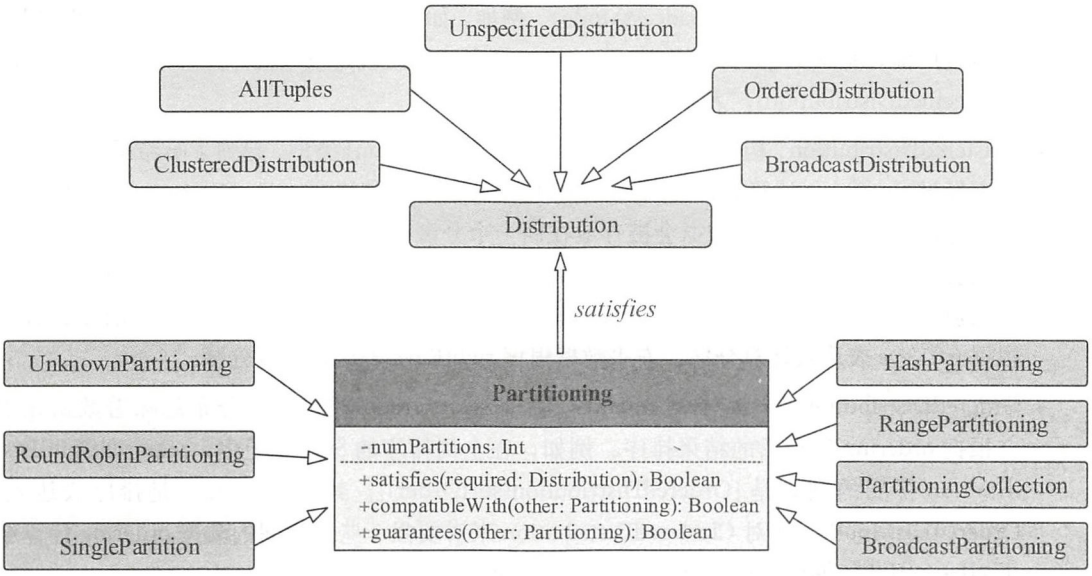


图 6.9 Partitioning 与 Distribution 具体实现的类

1. Distribution

Distribution 定义了查询执行时，同一个表达式下的不同数据元组（Tuple）在集群各个节点上的分布情况。

具体来讲，Distribution 可以用来描述以下两种不同粒度的数据特征。

（1）节点间（Inter-node）分区信息，即数据元组在集群不同的物理节点上是如何分区的。这个特性可以用来判断某些算子（例如 Aggregate）能否进行局部计算（Partial operation），避免全局操作的代价。

（2）分区数据内（Intra-partition）排序信息，即单个分区内数据是如何分布的。在 Spark 2.1 版本中包括以下 5 种 Distribution 的实现。

- UnspecifiedDistribution：未指定分布，无需确定数据元组之间的位置关系。
- AllTuples：只有一个分区，所有的数据元组存放在一起（Co-located）。例如，选取全局前 K 条数据的 GlobalLimit 算子，requiredChildDistribution 得到的列表就是 [AllTuples]，表示执行该算子需要全部的数据参与。
- BroadcastDistribution：广播分布，数据会被广播到所有节点上，构造参数 mode 为广播模式（BroadcastMode），广播模式可以为原始数据（IdentityBroadcastMode）或转换为 HashedRela-

tion 对象（HashedRelationBroadcastMode）。例如，如果是 Broadcast 类型的 Join 操作，假设左表做广播，那么 requiredChildDistribution 得到的列表就是 [BroadcastDistribution(mode), UnspecifiedDistribution]，表示左表为广播分布。

- ClusteredDistribution：构造参数 clustering 是 Seq[Expression] 类型，起到了哈希函数的效果，数据经过 clustering 计算后，相同 value 的数据元组会被存放在一起（Co-located）。如果有多个分区的情况，则相同数据会被存放在同一个分区中；如果只能是单个分区，则相同的数据会在分区内连续存放。例如，SortMerge 类型的 Join 操作中 requiredChildDistribution 列表就是 [ClusteredDistribution(leftKeys), ClusteredDistribution(rightKeys)]，表示左表数据根据 leftKeys 表达式计算分区，右表数据根据 rightKeys 表达式计算分区。
- OrderedDistribution：构造参数 ordering 是 Seq[SortOrder] 类型，该分布意味着数据元组会根据 ordering 计算后的结果排序。例如，在全局排序的 Sort 算子中，requiredChildDistribution 得到的列表是 [OrderedDistribution(sortOrder)]，其中 sortOrder 是排序表达式。OrderedDistribution 相对 ClusteredDistribution 来讲要强一些，相同的数据 ordering 计算结果相同，因此能够保持连续性并被划分到相同分区中。

2. Partitioning

Partitioning 定义了一个物理算子输出数据的分区方式，具体包括子 Partitionging 之间、目标 Partitioning 和 Distribution 之间的关系。

具体来讲，Partitioning 描述了 SparkPlan 中进行分区的操作，类似直接采用 API 进行 RDD 的 repartition 操作。根据图 6.9 的 UML 信息，Partitioning 接口中包含 1 个成员变量和 3 个函数来进行分区操作。

- numPartitions：指定该 SparkPlan 输出 RDD 的分区数目。
- satisfies(required: Distribution)：当前的 Partitioning 操作能否得到所需的数据分布（Required）。当不满足时（结果为 false），一般需要进行 repartition 操作，对数据进行重新组织。
- compatibleWith(other: Partitioning)：当存在多个子节点时，需要判断不同的子节点的分区操作是否兼容。直观地看，只有当两个 Partitioning 能够将相同 key 的数据分发到相同的分区时，才能够兼容。
- guarantees(other: Partitioning)：如果 A.gurantees(B) 能够为真，那么任何 A 进行分区操作所产生的数据行也能够被 B 产生。这样，B 就不需要再进行重分区操作。该方法主要用来避免冗余的重分区操作带来的性能代价。在默认情况下，一个 Partitioning 仅能够 gurantee

（保证）等于它本身的 Partitioning（相同的分区数目和相同的分区策略等）。

同样，Partitioning 接口的具体实现也有多种，如表 6.1 所示。

表 6.1 不同的 Partitioning 实现

分区方式	操作描述
UnknownPartitioning	不进行分区
RoundRobinPartitioning	在 1-numPartitions 范围内轮询式分区
HashPartitioning	基于哈希的分区方式
RangePartitioning	基于范围的分区方式
PartitioningCollection	分区方式的集合，描述物理算子的输出

这里以 HashPartitioning 为例，HashPartitioning 在 Spark SQL 中应用非常广泛，例如 Aggregation 和 Join 操作等。具体代码如下，构造参数 expressions 是用来计算 Hash 值的表达式列表，numPartitions 表示分区的数目。

```
case class HashPartitioning(expressions: Seq[Expression], numPartitions: Int)
  extends Expression with Partitioning with Unevaluable {
  override def satisfies(required: Distribution): Boolean = required match {
    case UnspecifiedDistribution => true
    case ClusteredDistribution(requiredClustering) =>
      expressions.forall(x => requiredClustering.exists(_.semanticEquals(x)))
    case _ => false
  }
  override def compatibleWith(other: Partitioning): Boolean = other match {
    case o: HashPartitioning => this.semanticEquals(o)
    case _ => false
  }
  override def guarantees(other: Partitioning): Boolean = other match {
    case o: HashPartitioning => this.semanticEquals(o)
    case _ => false
  }
}
```

根据其实现逻辑，只有当数据分布是 UnspecifiedDistribution，或者是 ClusteredDistribution 且包含 expressions 中所有的表达式时，satisfies 得到的结果才为 true。例如，数据表有 A 和 B 两列，ClusteredDistribution 是按照 [A, B] 计算得到的数据分布（requiredClustering 中的表达式包括 A 和 B），当前 HashPartitioning 根据 A 来计算（expression 中只包含 A），那么 satisfies 得到的结果为 true，意味着按照 A 进行哈希运算也能够保证相同的 A、B 在同一个分区。此外，compatibleWith 与 guarantees 也仅在 HashPartitioning 时，彼此之间才有效。

6.4.2 SparkPlan 的常用分区排序操作

作为抽象类，在 SparkPlan 默认实现中，将 outputPartitioning 设置为 UnknownPartitioning(0)，将 requiredChildDistribution 设置为 Seq[UnspecifiedDistribution]，且在数据有序性和排序操作方面不涉及任何动作。本小节对案例中涉及的几个 SparkPlan 的分区排序操作进行简要介绍。

（1）数据文件扫描执行算子（FileSourceScanExec）

```
case class FileSourceScanExec(@transient relation: HadoopFsRelation,
    output: Seq[Attribute], outputSchema: StructType, partitionFilters: Seq[Expression],
    dataFilters: Seq[Filter], override val metastoreTableIdentifier: Option[TableIdentifier])
  extends DataSourceScanExec {

  override val (outputPartitioning, outputOrdering): (Partitioning, Seq[SortOrder]) = {
    val bucketSpec = if (relation.sparkSession.sessionState.conf.bucketingEnabled) {
      relation.bucketSpec
    } else {
      None
    }
    bucketSpec match {
      case Some(spec) =>
        def toAttribute(colName: String): Option[Attribute] = output.find(_.name == colName)
        val bucketColumns = spec.bucketColumnNames.flatMap(n => toAttribute(n))
        if (bucketColumns.size == spec.bucketColumnNames.size) {
          val partitioning = HashPartitioning(bucketColumns, spec.numBuckets)
          val sortColumns =
           spec.sortColumnNames.map(x => toAttribute(x)).takeWhile(x => x.isDefined).map(_.get)
          val sortOrder = if (sortColumns.nonEmpty) {
            val files = selectedPartitions.flatMap(partition => partition.files)
            val bucketToFilesGrouping =
              files.map(_.getPath.getName).groupBy(file => BucketingUtils.getBucketId(file))
            val singleFilePartitions = bucketToFilesGrouping.forall(p => p._2.length <= 1)
            if (singleFilePartitions) {
              sortColumns.map(attribute => SortOrder(attribute, Ascending))
            } else {
              Nil
            }
          } else {
            Nil
          }
          (partitioning, sortOrder)
        } else {
          (UnknownPartitioning(0), Nil)
        }
      case _ => (UnknownPartitioning(0), Nil)
    }
  }
}
```

作为物理执行树中的叶子节点，FileSourceScanExec 中的分区排序信息会根据数据文件构造的初始的 RDD 进行设置。如果没有 bucket 信息，则分区与排序操作将分别为最简单的 UnknownPartitioning 与 Nil；当且仅当输入文件信息中满足特定的条件（代码中的 sortColumns 非空等）时，才会构造 HashPartitioning 与 SortOrder 类。

（2）过滤执行算子（FilterExec）与列剪裁执行算子（ProjectExec）

```
case class FilterExec(condition: Expression, child: SparkPlan)
  extends UnaryExecNode with CodegenSupport with PredicateHelper {
  override def outputOrdering: Seq[SortOrder] = child.outputOrdering
  override def outputPartitioning: Partitioning = child.outputPartitioning
}

case class ProjectExec(projectList: Seq[NamedExpression], child: SparkPlan)
  extends UnaryExecNode with CodegenSupport {
  override def outputOrdering: Seq[SortOrder] = child.outputOrdering
  override def outputPartitioning: Partitioning = child.outputPartitioning
}
```

由此可见，在过滤执行算子与列剪裁执行算子中，分区与排序的方式仍然沿用其子节点的方式，即不对 RDD 的分区与排序进行任何的重新操作。

通常情况下，LeafExecNode 类型的 SparkPlan 会根据数据源本身的特点（包括分块信息和数据有序性特征）构造 RDD 与对应的 Partitioning 和 Ordering 方式，UnaryExecNode 类型的 SparkPlan 大部分会沿用其子节点的 Partitioning 与 Ordering 方式（SortExec 等本身具有排序操作的执行算子例外），而 BinaryExecNode 往往会根据两个子节点的情况综合考虑，具体可以参见 SortMergeJoinExec 等执行算子的源码实现。SparkPlan 的 Partitioning 体系和 Exchange 节点息息相关，用于 SparkPlan 执行前的准备，这部分内容将在 6.6 节进行深入分析。

6.5 SparkPlan 生成

在 Spark SQL 中，当逻辑计划处理完毕后，会构造 SparkPlanner 并执行 plan() 方法对 LogicalPlan 进行处理，得到对应的物理计划。实际上，一个逻辑计划可能会对应多个物理计划，因此，SparkPlanner 得到的是一个物理计划的列表（Iterator[SparkPlan]）。

在讨论 SparkPlan 的生成逻辑之前，先看看 SparkPlanner 这个对象。如图 6.10 所示，SparkPlanner 继承自 SparkStrategies 类，而 SparkStrategies 类则继承自 QueryPlanner 基类，重要的 plan() 方法实现就在 QueryPlanner 类中。SparkStrategies 类本身不提供任何方法，而是在内部提供一批 SparkPlanner 会用到的各种策略（Strategy）实现。最后，在 SparkPlanner 层面将这些策略整合在一起，通过 plan() 方法进行逐个应用。

类似逻辑计划阶段的 Anaylzer 和 Optimizer，SparkPlanner 本身只是一个逻辑的驱动，各种

策略的 apply 方法把逻辑执行计划算子映射成物理执行计划算子。在 SparkPlanner 的调用逻辑和各种策略中，PlanLater 随处可见。根据其实现，PlanLater 本身也是 SparkPlan 的一种，区别在于 doExecute() 方法没有实现，表示不支持执行，所起到的作用仅仅是占位，等待后续步骤处理。

```
case class PlanLater(plan: LogicalPlan) extends LeafExecNode {
  override def output: Seq[Attribute] = plan.output
  protected override def doExecute(): RDD[InternalRow] = {
    throw new UnsupportedOperationException()
  }
}
```

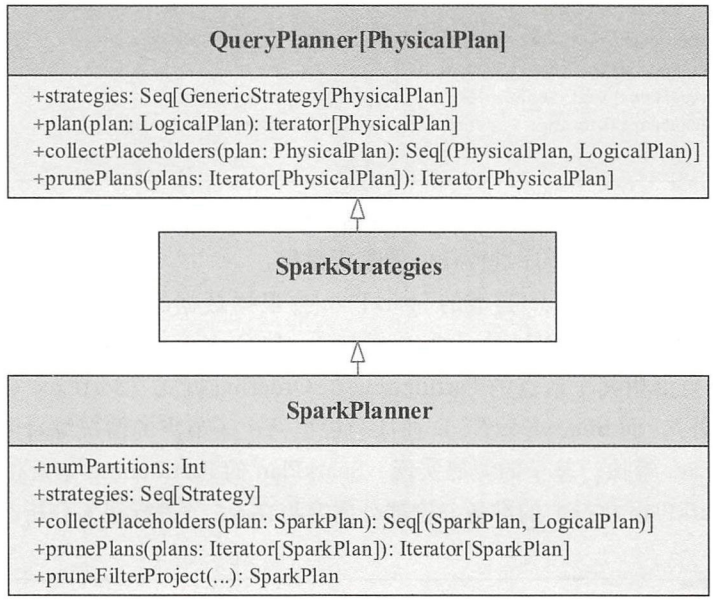


图 6.10　SparkPlanner 体系

生成物理计划的实现代码如下，plan() 方法传入 LogicalPlan 作为参数，将 strategies 应用到 LogicalPlan，生成物理计划候选集合（Candidates）。如果该集合中存在 PlanLater 类型的 SparkPlan，则通过 placeholder 中间变量取出对应的 LogicalPlan 后，递归调用 plan() 方法，将 PlanLater 替换为子节点的物理计划。最后，对物理计划列表进行过滤，去掉一些不够高效的物理计划。

```
def plan(plan: LogicalPlan): Iterator[PhysicalPlan] = {
    val candidates = strategies.iterator.flatMap(_(plan))
    val plans = candidates.flatMap { candidate =>
      val placeholders = collectPlaceholders(candidate)
      if (placeholders.isEmpty) {
```

```
        Iterator(candidate)
      } else {
        placeholders.iterator.foldLeft(Iterator(candidate)) {
          case (candidatesWithPlaceholders, (placeholder, logicalPlan)) =>
            val childPlans = this.plan(logicalPlan)
            candidatesWithPlaceholders.flatMap { candidateWithPlaceholders =>
              childPlans.map { childPlan =>
                candidateWithPlaceholders.transformUp {
                  case p if p == placeholder => childPlan
                }
              }
            }
        }
      }
    }
    val pruned = prunePlans(plans)
    assert(pruned.hasNext, s "No plan for $plan" )
    pruned
}
```

实际上，Spark SQL 在物理计划生成方面还有很多工作要做，例如，对生成的物理计划列表进行过滤筛选（prunePlans）在当前版本中并没有实现，生成多个物理计划后，仅仅是直接选取列表中的第一个作为最终结果（注：参见 QueryExecution 类中 sparkPlan 的生成代码）。有兴趣的读者可以在这方面进行深入研究。

6.5.1 物理计划 Strategy 体系

物理计划执行策略的构成如图 6.11 所示。所有的策略都继承自 GenericStrategy 类，其中定义了 planLater 和 apply 方法；SparkStrategy 继承自 GenericStrategy 类，对其中的 planLater 进行了实现，根据传入的 LogicalPlan 直接生成前述提到的 PlanLater 节点。此外，在 Spark SQL 中，Strategy 是 SparkStrategy 类的别名。

最后，各种具体的 Strategy 都实现了 apply 方法，将传入的 LogicalPlan 转换为 SparkPlan 的列表。如果当前的执行策略无法应用于该 LogicalPlan 节点，则返回的物理执行计划列表为空。因此，Strategy 是生成物理算子树的基础。

在实现上，各种 Strategy 会匹配传入的 LogicalPlan 节点，根据节点或节点组合的不同情形实行一对一的映射或多对一的映射。一对一的映射方式比较直观，以 BasicOperators 为例，该 Strategy 实现了各种基本操作的转换，其中列出了大量的映射关系，包括 Sort 对应 SortExec、Union 对应 UnionExec 等。多对一的情况涉及对多个 LogicalPlan 节点进行组合转换，这里称为逻辑算子树的模式匹配。目前在 Spark SQL 中，逻辑算子树的节点模式共有 4 种，如图 6.12 所示。

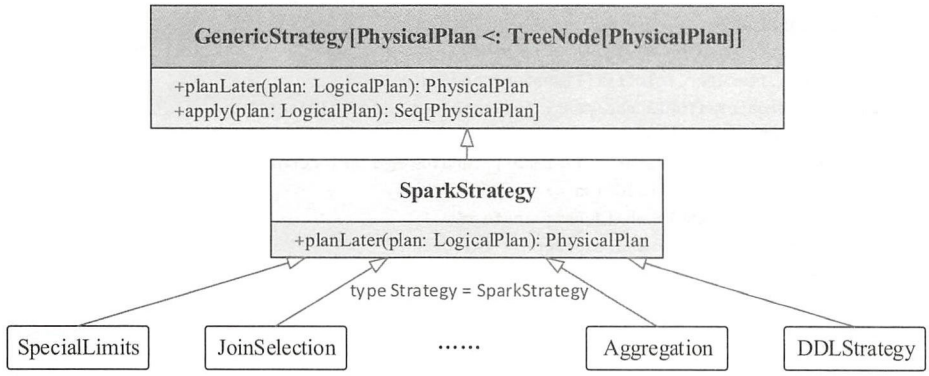


图 6.11　物理计划 Strategy 体系

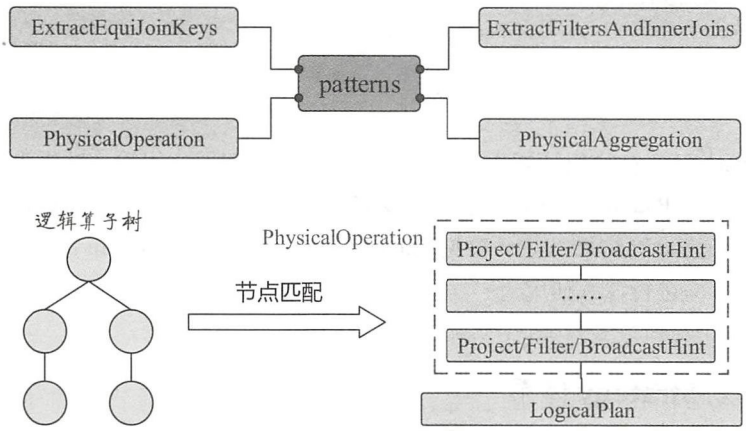


图 6.12　不同匹配模式

- ExtractEquiJoinKeys：针对具有相等条件的 Join 操作的算子集合，提取出其中的 Join 条件、左子节点和右子节点等信息。
- ExtractFiltersAndInnerJoins：收集 Inner 类型 Join 操作中的过滤条件，目前仅支持对左子树进行处理。
- PhysicalAggregation：针对聚合操作，提取出聚合算子中的各个部分，并对一些表达式进行初步的转换。
- PhysicalOperation：匹配逻辑算子树中的 Project 和 Filter 等节点，返回投影列、过滤条件集合和子节点。

图 6.12 中对 PhysicalOperation 模式进行了展示，如果匹配到 Project、Filter 或 BroadcastHint 3 种类型之一的 LogicalPlan 时，就会递归查找子节点。若子节点也是这 3 种类型之一，则收集

节点中的投影列或过滤条件。依此类推，直到碰到其他类型的 LogicalPlan 节点为止。剩下的 3 种模式都是关于 Aggregation 或 Join 操作的，这里先不做进一步的展开，详细的内容将在接下来的章节中有所涉及。

6.5.2 常见 Strategy 分析

在 SparkPlanner 中默认添加了 8 种 Strategy 来生成物理计划，如表 6.2 所示。FileSourceStrategy 与 DataSourceStrategy 主要针对数据源，Aggregation 与 JoinSelection 分别针对聚合与关联操作，BasicOperators 涉及范围最广，包含了过滤、投影等各种操作。本小节对一些常见的 Strategy 进行分析。

表 6.2 物理计划策略

物理计划生成策略	策略描述
FileSourceStrategy	数据文件扫描计划
DataSourceStrategy	各种数据源相关的计划
DDLStrategy	DDL 操作执行计划
SpecialLimits	特殊 limit 操作的执行计划
Aggregation	聚合算子相关的执行计划
JoinSelection	Join 操作相关的执行计划
InMemoryScans	内存数据表扫描计划
BasicOperators	对基本算子生成的执行计划

（1）文件数据源策略（FileSourceStrategy）：FileSourceStrategy 面向的是来自文件的数据源，主要的代码片段（中间逻辑涉及较多，详细内容参见代码）如下。该策略针对的典型模式是：能够匹配 PhysicalOperation 的节点集合加上 LogicalRelation 节点。在这种情况下，该策略会根据数据文件信息构建 FileSourceScanExec 这样的物理执行计划，并在此物理执行计划后添加过滤（FilterExec）与列剪裁（ProjectExec）物理计划。

Note：需要注意的是，即使在逻辑算子树上 LogicalRelation 节点往上存在多个过滤算子与投影算子，经过 PhysicalOperation 模式匹配，也会整合成为一个。

```
object FileSourceStrategy extends Strategy with Logging {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    case PhysicalOperation(projects, filters,
      l @ LogicalRelation(fsRelation: HadoopFsRelation, _, table)) =>
      ..........................................................................
      val scan = new FileSourceScanExec(fsRelation, outputAttributes, outputSchema,
                        partitionKeyFilters.toSeq, pushedDownFilters,table.map(_.identifier))
      val afterScanFilter = afterScanFilters.toSeq.reduceOption(expressions.And)
      val withFilter = afterScanFilter.map(execution.FilterExec(_, scan)).getOrElse(scan)
      val withProjections = if (projects == withFilter.output) {
```

```
        withFilter
      } else {
        execution.ProjectExec(projects, withFilter)
      }
      withProjections :: Nil
    case _ => Nil
  }
}
```

(2) 内存数据表扫描策略 (InMemoryScans): InMemoryScans 主要针对的是 InMemoryRelation 这个 LogicalPlan 节点，如以下代码所示，其逻辑同样是匹配 PhysicalOperation 这个模式，最终生成 InMemoryTableScanExec，并调用 SparkPlanner 中的 pruneFilterProject 方法对其进行过滤和列剪裁。

```
object InMemoryScans extends Strategy {
    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
      case PhysicalOperation(projectList, filters, mem: InMemoryRelation) =>
        pruneFilterProject(projectList, filters,
          identity[Seq[Expression]], // All filters still need to be evaluated.
          InMemoryTableScanExec(_, filters, mem)) :: Nil
      case _ => Nil
    }
}
```

(3) DDL 操作策略 (DDLStrategy)：DDLStrategy 在 Spark SQL 中仅针对 CreateTable 与 CreateTempViewUsing 这两种类型的节点，这两种情况都直接生成 ExecutedCommandExec 类型的物理计划。

(4) 基本操作策略 (BasicOperators)：BasicOperators 是专门针对各种基本操作类型的 LogicalPlan 节点，例如排序、过滤等，这种情况下，一般一对一地进行映射即可 (例如，Sort 逻辑节点映射为 SortExec 物理计划)。

基于上述介绍，案例所对应的物理计划的生成过程就非常简单了，如图 6.13 所示。Project 节点加上 Filter 节点对应 PhysicalOperation 模式，加上 LogicalRelation 节点，正好匹配到 FileSourceStrategy 策略。因此，整个转换逻辑都在 FileSourceStrategy 中完成，最终的物理计划包括 ProjectExec、FilterExec 和 FileSourceScanExec 共 3 个节点。

实际上，在 SparkPlanner 中最为复杂的策略是 Aggregation 和 JoinSelection，需要处理各种情况。上述案例中没有涉及这些操作，考虑到其复杂性，这两种策略会在后续章节中结合具体的案例进行分析。

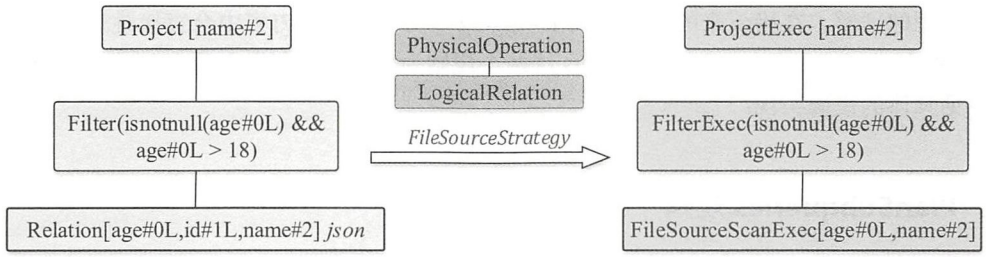


图 6.13 从 LogicalPlan 到 SparkPlan

6.6 执行前的准备

物理计划的生成意味着用户的 SQL 语句已经成功转换为 SparkPlan 物理算子树。然而，在通常情况下，到了这一步仍然不能直接提交给 Spark 系统执行。类似人工直接编写代码，应用提交前有必要从 Spark 系统本身的角度来考虑代码的正确性和高效性。因此，得到 SparkPlan 之后，还需要完成若干的准备工作，对树型结构的物理计划进行全局的整合处理或优化，具体实现参见如下代码。

```
lazy val executedPlan: SparkPlan = prepareForExecution(sparkPlan)

protected def prepareForExecution(plan: SparkPlan): SparkPlan = {
    preparations.foldLeft(plan) { case (sp, rule) => rule.apply(sp) }
}
```

在 QueryExection 中，最后阶段由 prepareforExecution 方法对传入的 SparkPlan 进行处理而生成 executedPlan，处理过程仍然基于若干规则（如表 6.3 所示），主要包括对 Python 中 UDF 的提取、子查询的计划生成等。

表 6.3 物理执行计划准备规则

执行准备规则	规则描述
python.ExtractPythonUDFs	提取 Python 中的 UDF 函数
PlanSubqueries	特殊子查询物理计划处理
EnsureRequirements	确保执行计划分区与排序正确性
CollapseCodegenStages	代码生成相关
ReuseExchange	Exchange 节点重用
ReuseSubquery	子查询重用

需要特别注意的是，CollapseCodegenStages 规则会根据 SparkPlan 的逻辑生成最终的 Java 执行代码。这部分机制非常复杂，将在 Tungsten 相关章节专门分析。考虑到代码生成不影响读者

对内部原理的理解，在接下来的两章（关于 Aggregation 和 Join）内容中会直接跳过该过程。本节以 PlanSubqueries 和 EnsureRequirements 这两个规则为例介绍执行前的准备工作，其他规则逻辑可参见具体的代码实现。

6.6.1 PlanSubqueries 规则

子查询是指嵌套在一个查询内部的完整查询，常见的子查询通常作为数据源出现在 SQL 的 From 关键字之后。Spark 2.0 版本及更高版本能够支持两种特殊情形的子查询，即 Scalar 类型和 Predicate 类型。

Scalar 类型的子查询返回单个值，具体又分为相关的（Correlated）类型和不相关的（Uncorrelated）类型。顾名思义，Uncorrelated 意味着子查询和主查询不存在相关性，Uncorrelated 类型的 Scalar 子查询对于所有的数据行都返回相同的值。因此，在主查询执行之前，Uncorrelated 子查询会首先执行。例如，计算每个学生的年龄和所有学生的最大年龄的 SQL 语句如下。

```
select name, age, (select max(age) from student) max_age from student
```

Correlated 类型的 Scalar 子查询意味着该子查询中包含了外层主查询中的相关属性，在 Spark SQL 中会等价转换为 Left Join 算子。Correlated Scalar 子查询在写法上有很多需要注意的地方，有兴趣的读者可以研究一下中间的转换过程。Predicate 类型的子查询表示子查询作为过滤谓词，在 Spark SQL 中可以出现在 EXISTS 和 IN 语句中。

PlanSubqueries 规则就是处理物理计划中的 ScalarSubquery 和 PredicateSubquery 这两种特殊的子查询，具体实现如以下代码所示，遍历物理算子树中的所有表达式，碰到 ScalarSubquery 或 PredicateSubquery 表达式时，进入子查询中的逻辑，递归得到子查询的物理执行计划（executedPlan），然后封装为 ScalarSubquery 和 InSubquery 表达式。

```
case class PlanSubqueries(sparkSession: SparkSession) extends Rule[SparkPlan] {
  def apply(plan: SparkPlan): SparkPlan = {
    plan.transformAllExpressions {
      case subquery: expressions.ScalarSubquery =>
        val executedPlan = new QueryExecution(sparkSession, subquery.plan).executedPlan
        ScalarSubquery(SubqueryExec(s"subquery${subquery.exprId.id}", executedPlan), subquery.exprId)
      case expressions.PredicateSubquery(query, Seq(e: Expression), _, exprId) =>
        val executedPlan = new QueryExecution(sparkSession, query).executedPlan
        InSubquery(e, SubqueryExec(s"subquery${exprId.id}", executedPlan), exprId)
    }
  }
}
```

6.6.2 EnsureRequirements 规则

顾名思义，EnsureRequirements 用来确保物理计划能够执行所需要的前提条件，包括对分区和排序逻辑的处理。前面章节已经介绍过，在特定情形下，SparkPlan 对输入数据的分布（Distribution）情况和排序（Ordering）特性有着一定的要求。例如，SortMerge 类型的 Join 算子，要求输入数据已经按照 Hash 方式分区且处于有序状态。

如果输入数据的分布或有序性无法满足当前节点的处理逻辑，则 EnsureRequirements 规则会在物理计划中添加一些 Shuffle 操作或排序操作来达到要求，体现在物理算子树上就是加入 Exchange 节点或 SortExec 节点。此外，该过程还涉及依赖信息（ShuffleDependcency）的创建、ShuffledRowRDD 的构造等，其处理逻辑可以算是整个物理计划阶段最为复杂的部分。

从 EnsureRequirements 规则的 apply 方法可知，在遍历 SparkPlan 的过程中，当匹配到 Exchange 节点（ShuffleExchange）且其子节点也是 Exchange 类型时，会检查两者的 Partitioning 方式，判断能否消除多余的 Exchange 节点。除此情况外，遍历过程中会逐个调用 ensureDistributionAndOrdering 方法来确保每个节点的分区与排序需求。因此，EnsureRequirements 规则的核心逻辑体现在 ensureDistributionAndOrdering 方法中，可以将其大致过程分为以下 3 步。

（1）添加 **Exchange** 节点

Exchange 本身也是 UnaryExecNode 类型的 SparkPlan，在 Spark SQL 中被定义为抽象类，如图 6.14 所示。继承 Exchange 的子类有 BroadcastExchangeExec 和 ShuffleExchange 两种。很明显，ShuffleExchange 会通过 Shuffle 操作进行重分区处理，而 BroadcastExchangeExec 则对应广播操作。

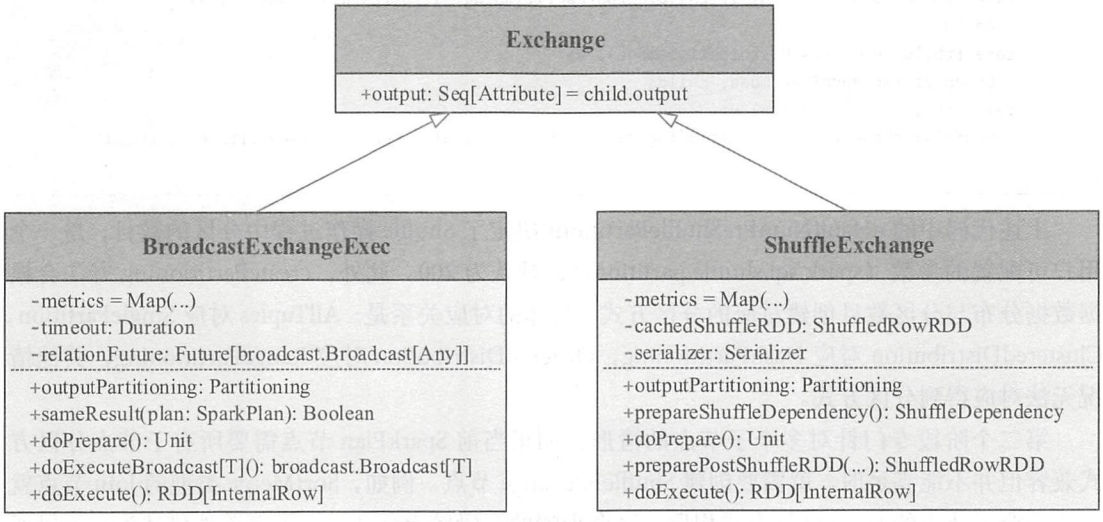


图 6.14 不同的 Exchange 类型

Exchange 节点是实现数据并行化的重要算子，用于解决数据分布（Distribution）相关问题。具体来讲，需要添加 Exchange 节点的情形有以下两种。

- 数据分布不满足：子节点的物理计划输出数据无法满足（Satisfies）当前物理计划处理逻辑中对数据分布的要求，例如子节点输出数据分布为 UnspecifiedDistribution，而当前物理计划对输入数据分布的需求是 OrderedDistribution。
- 数据分布不兼容：当前物理计划为 BinaryExecNode 类型，即存在两个子物理计划时，两个子物理计划的输出数据可能不兼容（Compatile）。例如，Hash 的方式不同，导致应该在同一个分区的数据最终落到不同的节点上。在这种情况下，也需要创建 Exchange 节点重新进行 Shuffle 操作。

在 ensureDistributionAndOrdering 方法中，添加 Exchange 节点的过程可以细分为两个阶段，分别针对单个节点和多个节点。第一个阶段是判断每个子节点的分区方式是否可以满足（Satisfies）对应所需的数据分布。如果满足，则不需要创建 Exchange 节点；否则根据是否广播来决定添加何种类型的 Exchange 节点，具体实现逻辑如以下代码所示。例如，SortMerge 类型的 Join 节点中 requiredChildDistribution 列表为 [ClusteredDistribution(leftKeys), ClusteredDistribution(rightKeys)]，假设两个子节点的 Partitioning 都无法输出该数据分布，那么就会添加两个 ShuffleExchange 节点。

```
children = children.zip(requiredChildDistributions).map {
      case (child, distribution) if child.outputPartitioning.satisfies(distribution) =>
        child
      case (child, BroadcastDistribution(mode)) =>
        BroadcastExchangeExec(mode, child)
      case (child, distribution) =>
        ShuffleExchange(createPartitioning(distribution, defaultNumPreShufflePartitions), child)
}
```

上述代码中的 defaultNumPreShufflePartitions 决定了 Shuffle 操作过程中分区的数目，是一个用户可配置的参数（spark.sql.shuffle.partitions），默认为 200。此外，createPartitioning 方法会根据数据分布与分区数目创建对应的分区方式，具体的对应关系是：AllTuples 对应 SinglePartition，ClusteredDistribution 对应 HashPartitioning，OrderedDistribution 对应 RangePartitioning，其他情况无法对应得到分区方式。

第二个阶段专门针对多个子节点的情形，如果当前 SparkPlan 节点需要所有子节点分区方式兼容但并不能满足时，就需要创建 ShuffleExchange 节点。例如，SortMerge 类型的 Join 节点就需要两个子节点的 Hash 计算方式相同。这个步骤的逻辑较为复杂，可以简单地描述为：如果所有的子节点 outputPartitioning 能够保证由最大分区数目创建新的 Partitioning，则子节点输出的数据并不需要重新 Shuffle，那么只需要使用已有的 outputPartitioning 方式即可，没有必要重新

创建新的 Exchange 节点；否则，至少有一个子节点的输出数据需要重新进行 Shuffle 操作。重分区的数目（NumPartitions）根据是否所有的子节点输出都需要 Shuffle 来判断，若是，则采用默认的 Shuffle 分区配置数目；否则，取子节点中最大的分区数目。

```
if (children.length > 1 && requiredChildDistributions.exists(requireCompatiblePartitioning)
        && !Partitioning.allCompatible(children.map(_.outputPartitioning))) {
      val maxChildrenNumPartitions = children.map(_.outputPartitioning.numPartitions).max
      val useExistingPartitioning = children.zip(requiredChildDistributions).forall {..}
      children = if (useExistingPartitioning) {
        children
      } else {
        val numPartitions = {...}
        children.zip(requiredChildDistributions).map {
          case (child, distribution) =>
            val targetPartitioning = createPartitioning(distribution, numPartitions)
            if (child.outputPartitioning.guarantees(targetPartitioning)) {
              child
            } else {
              child match {
                case ShuffleExchange(_, c, _) => ShuffleExchange(targetPartitioning, c)
                case _ => ShuffleExchange(targetPartitioning, child)
              }
          }
        }
      }
}
```

第二个阶段的最后会根据创建的 Partitioning 对当前 SparkPlan 节点进行操作。因为在第一个阶段针对单个子节点进行处理时有可能已经创建了 ShuffleExchange 节点，那么这种情况下会对其进行替换，其他情况下直接创建新的 ShuffleExchange 节点即可。

（2）应用 **ExchangeCoordinator** 协调分区

根据源码中的描述，ExchangeCoordinator 用来确定物理计划生成的 Stage 之间如何进行 Shuffle 的行为。顾名思义，其作用在于协助 ShuffleExchange 节点“更好地”执行。在 Spark 2.1 版本中，ExchangeCoordinator 的功能比较简单，仅用于确定数据 Shuffle 后（Post-shuffle）的分区数目。

实际上，这个需求在 Spark SQL 生产环境中是非常迫切的，Shuffle 后的分区数目如果设置得太大，则会相应地分配计算单元，带来调度压力且导致资源浪费；如果设置得太小，则单个分区数据量大，处理起来性能低，甚至导致 OOM 等问题出现。作为公共的平台，能够根据数据特点自动计算合理的分区数目是一个必不可少的能力，ShuffleExchange 中默认的固定设置显然无法满足该需求。

ExchangeCoordinator 是 ShuffleExchange 的 Option 类型构造参数，属于“可有可无”的，默认情况下不会被创建。ExchangeCoordinator 针对的是一批 SparkPlan 节点，根据 withExchange-

Coordinator 方法的逻辑，需要满足两个条件。

- Spark SQL 的自适应机制开启，对应的参数（spark.sql.adaptive.enabled）设置为 true。
- 这批 SparkPlan 节点能够支持协调器，一种情况是至少存在一个 ShuffleExchange 类型的节点且所有节点的输出分区方式都是 HashPartitioning，另一种情况是节点数目大于 1 且每个节点输出数据的分布都是 ClusteredDistribution 类型。

需要注意的是，当 ShuffleExchange 中加入了 ExchangeCoordinator 来协调分区数目时，需要知道子物理计划输出的数据统计信息，因此在协调之前需要将 ShuffleExchange 之前的 Stage 提交到集群执行来获取相关信息（参考 doEstimationIfNecessary 方法）。

（3）添加 **SortExec** 节点

排序的处理在分区处理（创建完 Exchange）之后，其逻辑相对简单，不用考虑子节点彼此之间的兼容问题，只需要对每个子节点单独处理，ensureDistributionAndOrdering 方法中的逻辑可参见如下代码片段。

```
children = children.zip(requiredChildOrderings).map {case (child, requiredOrdering) =>
  if (requiredOrdering.nonEmpty) {
   val orderingMatched = if (requiredOrdering.length > child.outputOrdering.length) {false} else {
      requiredOrdering.zip(child.outputOrdering).forall {
         case (requiredOrder, childOutputOrder) => requiredOrder.semanticEquals(childOutputOrder)
      }
   }
   if (!orderingMatched) {SortExec(requiredOrdering, global = false, child = child)} else {child}
  } else {
      child
  }
}
```

可以看到，当且仅当所有子节点的输出数据的排序信息满足当前节点所需时，才不需要添加 SortExec 节点；否则，需要在当前节点上添加 SortExec 为父节点。至此，Ensurerequirements 规则的处理逻辑结束，调用 TreeNode 中的 withNewChildren 将 SparkPlan 中原有的子节点替换为新的子节点。

ShuffleExchange 执行得到的 RDD 称为 ShuffledRowRDD，这里介绍一下生成过程。概括起来，ShuffleExchange 执行 doExecute 时，首先会创建 ShuffleDependency，然后根据 ShuffleDependency 构造 ShuffledRowRDD，其中的重点在于 ShuffleDependency 的创建，分为以下两种情况。

- 包含 ExchangeCoordinator 协调器：如果需要 ExchangeCoordinator 协调 ShuffledRowRDD 的分区，则需要先提交该 ShuffleExchange 之前的 Stage 到 Spark 集群执行，完成之后确定 ShuffledRowRDD 的分区索引等信息。

- 直接创建：直接执行 prepareShuffleDependency 方法来创建 RDD 的依赖，然后根据 ShuffleDependency 创建 ShuffledRowRDD 对象。在这种情况下，ShuffledRowRDD 中每个分区的 ID 与 Exchange 节点进行 Shuffle 操作后的数据分区是一对一的映射关系。

直接创建 ShuffleDependency 的过程中，首先会根据传入的 newPartitioning 信息构造对应的 partitioner，然后基于该 partitioner 生成 ShuffleDependency 最重要的构造参数 rddWithPartitionIds（RDD[Product2[Int, InternalRow]] 类型），其中 [Int, InternalRow] 代表某行数据及其所在分区的 partitionId，其取值范围为 [0, numPartitions-1]。最终在 rddWithPartitionIds 基础上创建 ShuffleDependency 对象。

6.7 本章小结

在 Spark SQL 中，物理计划阶段是整个流程的最后一步。在此阶段，逻辑算子树最终会被转换为可以提交到 Spark 系统执行的物理算子树。从实现层面看，物理计划阶段已经脱离了 Catalyst 框架，与底层的执行平台紧密相关，因此这个阶段会涉及 Spark 系统各个方面的技术点。

本章的目的在于从全局的视角加深读者对 SparkPlan 阶段的理解。类似于第 5 章，本章的内容也包含两方面，一方面是横向分析，包括 SparkPlan 本身的各种体系，以及最终生成可执行物理计划所经过的 3 个通用阶段；另一方面仍然结合案例，从纵向角度详细分析每一步的转换过程。

第7章 Spark SQL 之 Aggregation 实现

聚合操作（Aggregation）指的是在原始数据的基础上按照一定的逻辑进行整合，从而得到新的数据，一般通过聚合函数（如 count、max 和 sum 等）汇总多行的信息。聚合查询一直以来都是数据分析应用中必不可少的部分，在各种 SQL 算子中占据着重要地位，本章对 Spark SQL 实现聚合查询的内部机制进行详细分析。

7.1 Aggregation 执行概述

一条 SQL 语句的执行会经历逻辑计划和物理计划的各个阶段，同样的，聚合查询也不例外。本节从全局视角出发，纵向考察 Aggregation 执行的整个流程（从 SQL 语句文法定义到物理执行），并在后续小节中对其中的重要技术点进行分析。

7.1.1 文法定义

在 Catalyst 的 SqlBase.g4 文法文件中，聚合语句 aggregation 定义如下：在常见的聚合查询中，通常包括分组语句（group by）和聚合函数（aggregate function）；聚合函数出现在 Select 语句中的情形较多，定义在 functionCall 标签的 primaryExpression 表达式文法中，qualifiedName 对应函数名，括号内部是该函数的参数列表。

```
primaryExpression: qualifiedName '(' (setQuantifier? expression (',' expression)*)? ')' (OVER
    windowSpec)? #functionCall

aggregation: GROUP BY groupingExpressions+=expression (',' groupingExpressions+=expression)*
             (WITH kind=ROLLUP
             | WITH kind=CUBE
             | kind=GROUPING SETS '(' groupingSet (',' groupingSet)* ')')?
```

从上述文法定义中可以看到，完整的聚合查询的关键字包括 group by、cube、grouping sets 和 rollup 4 种。分组语句 group by 后面可以是一个或多个分组表达式（groupingExpressions）。除

简单的分组操作外，聚合查询还支持 OLAP 场景下的多维分析，包括 rollup、cube 和 grouping sets 3 种操作。

为了分析方便，本章以一个简单聚合查询实例作为研究对象。数据仍然使用第 3 章中创建的 student 关系表，在查询语句中加入聚合操作：按照表中的 id 列分组，并对每个分组的数据使用 count 聚合函数计算数据条数。

```
select id, count(name) from student group by id
```

上述聚合查询语句生成的语法树如图 7.1 所示。相对于第 3 章中的非聚合查询，该语法树除 id 列外，还有对 name 的 count 操作所产生的新列，因此 NamedExpressionSeqContext 节点包含两个子节点。此外，代表数据表信息的 FromClauseContext 子树没有变化，仍然是 QuerySpecificationContext 节点的第 2 个子节点。

图 7.1 聚合查询抽象语法树

加入聚合操作后的语法树最重要的元素是 FunctionCallContext 节点和 AggregationContext 节点。AggregationContext 节点反映在语法树中即图 7.1 中 QuerySpecificationContext 节点下的第 3 个子节点，其子节点（从 ExpressionContext 一直到 ColumnReferenceContext）对应 group by 语句后面的 id 列。用来表示聚合函数的 FunctionCallContext 节点的结构比较好理解，其子节点 QualifiedNameContext 代表函数名，ExpressionContext 表示函数的参数表达式（对应 SQL 语句中的 name 列）。

7.1.2 聚合语句 Unresolved LogicalPlan 生成

从上述抽象语法树生成 Unresolved LogicalPlan 是接下来的重要一步，该过程主要由 AstBuilder 完成。在分析 AstBuilder 具体动作之前，先简要了解一下后续聚合操作会用到的 Aggregate 逻辑算子树节点。

在 Spark SQL 中，Aggregate 逻辑算子树节点是 UnaryNode 中的一种，属于基本的逻辑算子（basicLogicalOperator）。如图 7.2 所示，该逻辑算子树节点通过分组表达式列表（groupingExpressions）、聚合表达式列表（aggregateExpressions）和子节点（child）构造而成，其中分组表达式类型都是 Expression，而聚合表达式类型都是 NamedExpression，意味着聚合表达式一般都需要设置名字。同时，Aggregate 的输出函数 output 对应聚合表达式列表中的所有属性值。判断一个聚合算子是否已经被解析过（resolved=true）需要满足 3 个条件。

- 该算子中的所有表达式都已经被解析过了。
- 其子节点已经被解析过了。
- 该节点中不包含窗口（Window）函数表达式。

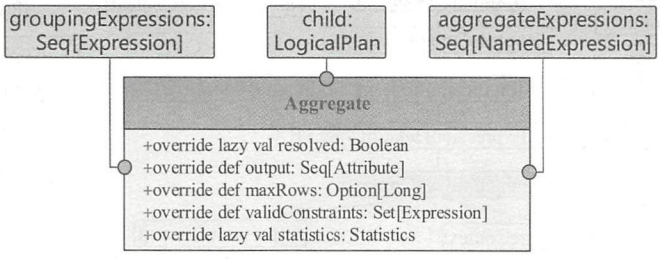


图 7.2　逻辑算子树聚合算子

如图 7.3 所示，上述聚合查询从抽象语法树生成 Unresolved LogicalPlan 主要涉及以下 3 个函数的调用。

- 针对 QuerySpecificationContext 节点，执行 visitQuerySpecification，会先后调用 visitFromClause 和 withQuerySpecification 函数。

- 在 visitFromClause 函数中，针对 FromClauseContext 节点生成 UnresolvedRelation 逻辑算子节点，对应数据表。
- 在返回的 UnresolvedRelation 节点上，执行 withQuerySpecification 函数，实际上这里具体执行的是 withAggregation 函数，在 UnresolveRelation 节点上生成 Aggregate 逻辑算子树节点，返回完整的逻辑算子树。

对于上述 Unresolved LogicalPlan 的生成过程，读者有必要关注一下 visitFunctionCall 这个方法。visitFunctionCall 方法由 visitNamedExpression 调用，属于 Select 语句中的一部分，用来生成聚合函数对应的表达式。最终得到的 UnresolvedFunction 包含了聚合函数名、参数列表和是否包含 distinct 的布尔值。

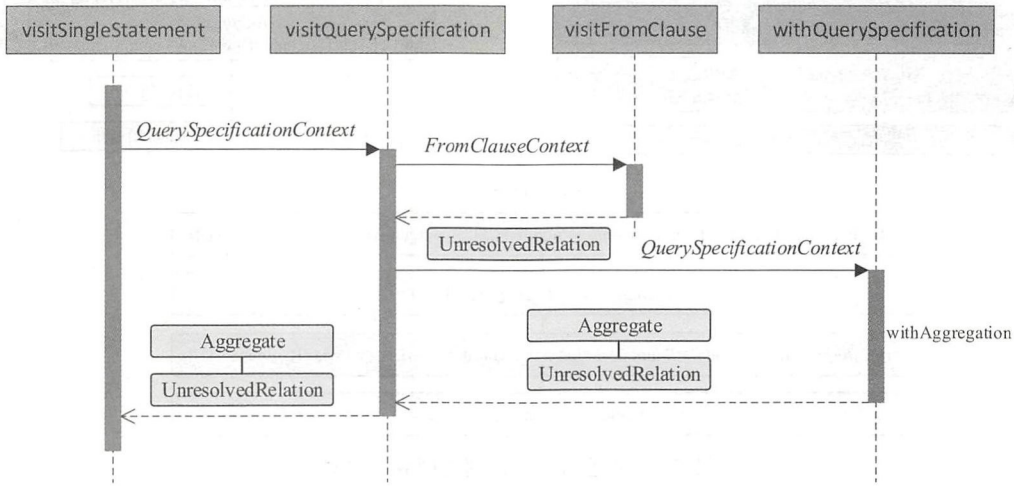


图 7.3 逻辑算子树生成

7.1.3 从逻辑算子树到物理算子树

生成 Unresolved LogicalPlan 之后，就是逻辑算子树到物理算子树的各个阶段。上述聚合语句同样涉及 Analyzed LogicalPlan、Optimized LogicalPlan、SparkPlan 和 ExecutedPlan 多个步骤，如图 7.4 所示。由于前面章节已经对这些步骤中的细节详细阐述过，所以本小节仅对与聚合查询相关的一些规则进行分析。

从 Unresolved LogicalPlan 到 Analyzed LogicalPlan 经过了 4 条规则的处理。对于聚合查询来说，比较重要的是其中的 ResolveFunctions 规则，用来分析聚合函数。对于 UnresolvedFunction 表达式，Analyzer 会根据函数名和函数参数去 SessionCatalog 中查找，而 SessionCatalog 会根

据 FunctionRegistry 中已经注册的函数信息得到对应的聚合函数（AggregateFunction）。值得注意的是，在 FunctionRegistry 中定义了 expression[T <: Expression](name: String)(implicit tag: ClassTag[T]) 函数，与 HashMap 一样，对传入的 Seq[Expression] 函数参数列表进行判断，如果查找到的函数接受的是多个 Expression 参数，则参数列表会拆分为多个 Expression 参数来完成对 AggregateFunction 的构造，最后返回给 SessionCatalog。

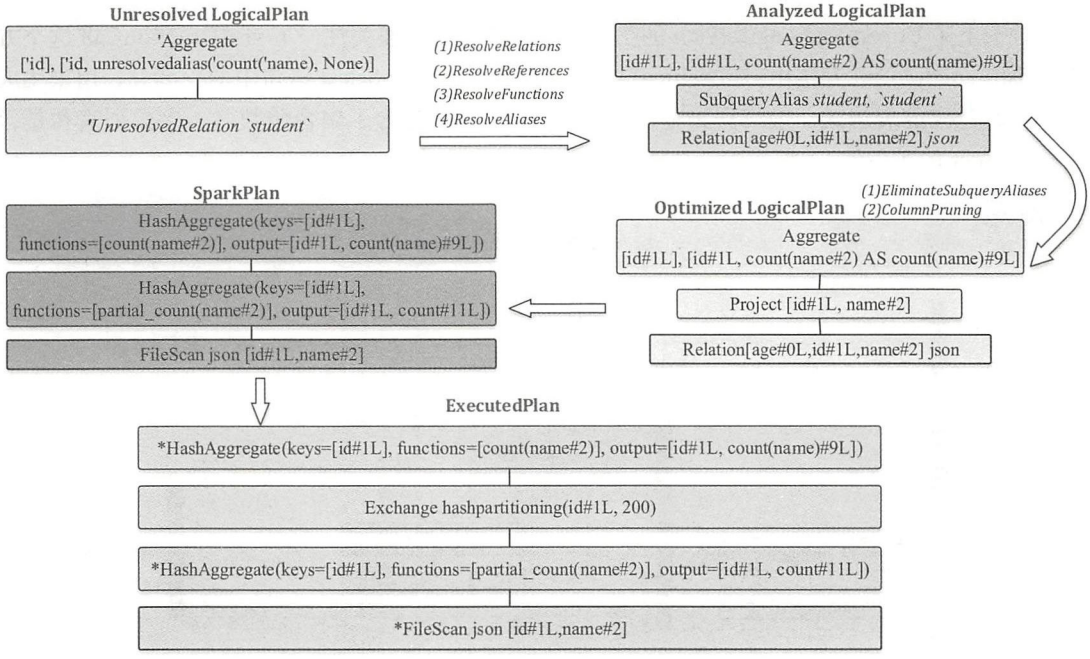


图 7.4　从逻辑算子树到物理算子树概览

从 Analyzed LogicalPlan 到 Optimized LogicalPlan 分别经过了别名消除（EliminateSubqueryAliases）规则与列剪裁（ColumnPruning）规则的处理，这里不再赘述。而从 Optimized LogcialPlan 到物理执行计划 SparkPlan 进行转换时，主要经过了 FileSourceStrategy 和 Aggregation 这两个策略（Strategy）的处理。FileSourceStrategy 会应用到 Project 和 Relation 节点，匹配过程在第 6 章已经分析过，在此不再赘述。

```
object PhysicalAggregation {
  type ReturnType =
    (Seq[NamedExpression], Seq[AggregateExpression], Seq[NamedExpression], LogicalPlan)
  def unapply(a: Any): Option[ReturnType] = a match {
    case logical.Aggregate(groupingExpressions, resultExpressions, child) =>
      val aggregateExpressions = resultExpressions.flatMap { expr =>
        expr.collect { case agg: AggregateExpression => agg }
      }.distinct
```

```
      val namedGroupingExpressions = groupingExpressions.map {
        case ne: NamedExpression => ne -> ne
        case other =>
          val withAlias = Alias(other, other.toString)()
          other -> withAlias
      }
      val groupExpressionMap = namedGroupingExpressions.toMap
      val rewrittenResultExpressions = resultExpressions.map { expr =>
        expr.transformDown {
          case ae: AggregateExpression => ae.resultAttribute
          case expression =>
            groupExpressionMap.collectFirst {
              case (expr, ne) if expr semanticEquals expression => ne.toAttribute
            }.getOrElse(expression)
        }.asInstanceOf[NamedExpression]
      }
      Some((namedGroupingExpressions.map(_._2), aggregateExpressions,
        rewrittenResultExpressions, child))
    case _ => None
  }
}
```

对于聚合查询，逻辑算子树转换为物理算子树，必不可少的是 Aggregation 转换策略。实际上，Aggregation 策略是基于 PhysicalAggregation 的。如图 7.5 所示，与 PhysicalOperation 类似，PhysicalAggregation 也是一种逻辑算子树的模式，用来匹配逻辑算子树中的 Aggregate 节点并提取该节点中的相关信息。PhysicalAggregation 在提取信息时会进行以下转换。

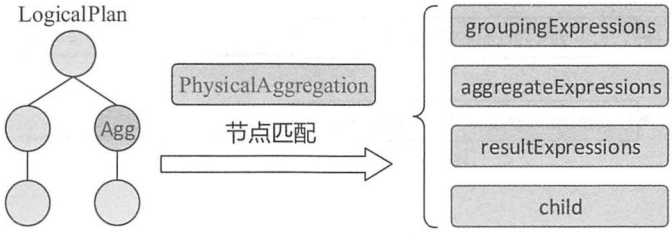


图 7.5 聚合算子匹配模式

- 去重：对 Aggregate 逻辑算子节点中多次重复出现的聚合操作进行去重，参见 PhysicalAggregation 中 aggregateExpressions 表达式的逻辑，收集 resultExpressions 中的聚合函数表达式，然后执行 distinct 操作。
- 命名：参见上述代码中 namedGroupingExpressions 的操作，对未命名的分组表达式（Grouping expressions）进行命名（套上一个 Alias 表达式），这样方便在后续聚合过程中进行引用。

- 分离：对应 rewrittenResultExpressions 中的操作逻辑，从最后结果中分离出聚合计算本身的值，例如“count+1”会被拆分为 count（AggregateExpression）和“count.resultAttribute + 1”的最终计算。

经过上述处理，PhysicalAggregation 模式返回的聚合操作相关表达式如表 7.1 所示，其中 aggregateExpressions 对应聚合函数，而 resultExpressions 则包含了 Select 语句中选择的所有列信息。

表 7.1　PhysicalAggregation 匹配模式表达式生成

访问操作	返回的 Expression
groupingExpressions	[id#1L]
aggregateExpressions	[count(name#2)]
resultExpressions	[id#1L, count(name#2)#8L AS count(name)#9L]

得到上述各种聚合信息之后，Aggregation 策略会根据这些信息生成相应的物理计划。如图 7.6 所示，不同情况下生成的物理计划也不相同。当聚合表达式中存在不支持 Partial 方式且不包含 Distinct 函数时，调用的是 planAggregateWithoutPartial 方法；当聚合表达式都支持 Partial 方式且不包含 Distinct 函数时，调用的是 planAggregateWithoutDistinct 方法；当聚合表达式都支持 Partial 方式且存在 Distinct 函数时，调用的是 planAggregateWithOneDistinct 方法。

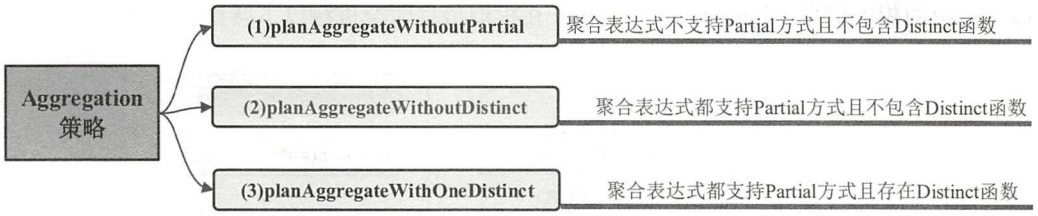


图 7.6　聚合算子生成策略

这里的 Partial 方式表示聚合函数的模式，能够支持预先局部聚合，这方面的内容会在下一节详细介绍。对应实例中的聚合语句，因为 count 函数支持 Partial 方式，因此调用的是 planAggregateWithoutDistinct 方法，生成了图 7.4 中的两个 HashAggregate（聚合执行方式中的一种，后续详细介绍）物理算子树节点，分别进行局部聚合与最终的聚合。最后，在生成的 SparkPlan 中添加 Exchange 节点，统一排序与分区信息，生成物理执行计划（ExecutedPlan）。

本章后续内容安排如下：在 7.2 节中会对聚合函数进行详细分析，包括聚合函数缓冲区和不同的聚合模式介绍，以及聚合函数的各种类型；7.3 节重点分析聚合执行的技术原理；7.4 节介绍窗口函数实现机制；7.5 节是 OLAP 场景的多维分析实现；最后在 7.6 节进行全章内容的总结。

7.2 聚合函数（AggregateFunction）

聚合函数（AggregateFunction）是聚合查询中非常重要的元素。在实现上，聚合函数是表达式中的一种，和 Catalyste 中定义的聚合表达式（AggregationExpression）紧密关联。无论是在逻辑算子树还是物理算子树中，聚合函数都是以聚合表达式的形式进行封装的，同时聚合函数表达式中也定义了直接生成聚合表达式的方法。

聚合表达式（AggregationExpression）的成员变量和函数如图 7.7 所示，根据命名，这些变量和函数的含义很好理解。例如，resultAttribute 表示聚合结果，获取子节点的 children 方法并返回聚合函数表达式；dataType 函数直接调用聚合函数中的 dataType 函数获取数据类型。在默认情况下，聚合表达式的 foldable 函数返回的是 false，因为聚合表达式一般无法静态得到最终的结果，需要经过进一步的计算。

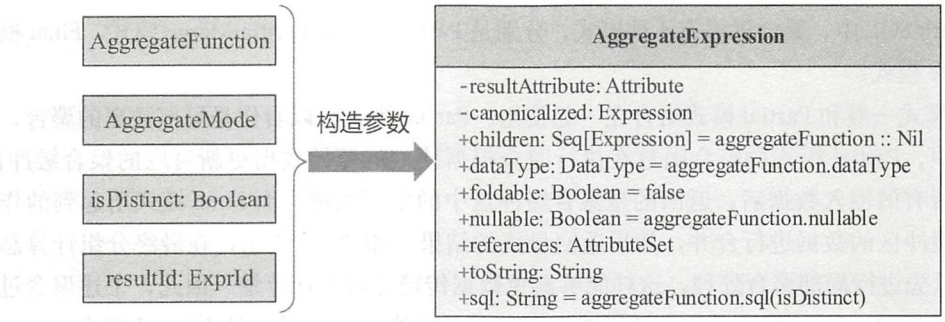


图 7.7 Aggregation 表达式

7.2.1 聚合缓冲区与聚合模式（AggregateMode）

聚合查询在计算聚合值的过程中，通常都需要保存相关的中间计算结果，例如 max 函数需要保存当前最大值，count 函数需要保存当前的数据总数，求平均值的 avg 函数需要同时保存 count 和 sum 的值，更复杂的函数（如 pencentil 等）甚至需要临时存储全部的数据。聚合查询计算过程中产生的这些中间结果会临时保存在聚合函数缓冲区。

1. 聚合函数缓冲区

聚合函数缓冲区是指在同一个分组的数据聚合的过程中，用来保存聚合函数计算中间结果的内存空间。

聚合函数缓冲区的定义有一个前提条件，即聚合函数缓冲区针对的是处于同一个分组内(实例中属于同一个 id) 的数据。需要注意的是，查询中可能包含多个聚合函数，因此聚合函数缓冲区是多个聚合函数所共享的。

在聚合函数的定义中，与聚合缓冲区相关的基本信息包括：聚合缓冲区的 Schema 信息(aggBufferSchema)，返回为 StructType 类型；聚合缓冲区的数据列信息 (aggBufferAttributes)，返回的是 AttributeReference 列表 (Seq[AttributeReference])，对应缓冲区数据列名。显然，聚合函数缓冲区中的值会随着数据处理而不断进行更新，因此该缓冲区是可变的 (Mutable)。此外，当聚合函数处理新的数据行时，需要知道该数据行的列构成信息，在 AggregateFunction 中也定义了 inputAggBufferAttributes 函数来获得输入数据的组成情况。通常情况下，inputAggBuffereAttributes 返回的都是自动从 aggBufferAttributes 获得的结果。

2. 聚合模式

在 Spark SQL 中，聚合过程有 4 种模式，分别是 Partial 模式、ParitialMerge 模式、Final 模式和 Complete 模式。

Final 模式一般和 Partial 模式组合在一起使用。Partial 模式可以看作是局部数据的聚合，在具体实现中，Partial 模式的聚合函数在执行时会根据读入的原始数据更新对应的聚合缓冲区，当处理完所有的输入数据后，返回的是聚合缓冲区中的中间数据。而 Final 模式所起到的作用是将聚合缓冲区的数据进行合并，然后返回最终的结果。如图 7.8 所示，在最终分组计算总和之前，可以先进行局部聚合处理，这样能够避免数据传输并减少计算量。因此，上述聚合过程中在 map 阶段的 sum 函数处于 Partial 模式，在 reduce 阶段的 sum 函数处于 Final 模式。

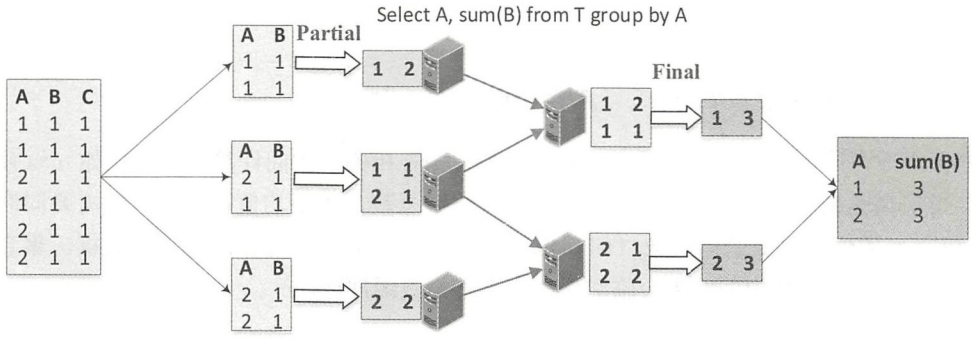


图 7.8　Partial 与 Final 聚合模式

Complete 模式和上述的 Partial/Final 组合方式不一样，不进行局部聚合计算。图 7.9 展示了同样的聚合函数采用 Complete 模式的情形。可以看到，最终阶段直接针对原始输入，中间没有

局部聚合过程。一般来讲，Complete 模式应用在不支持 Partial 模式的聚合函数中。

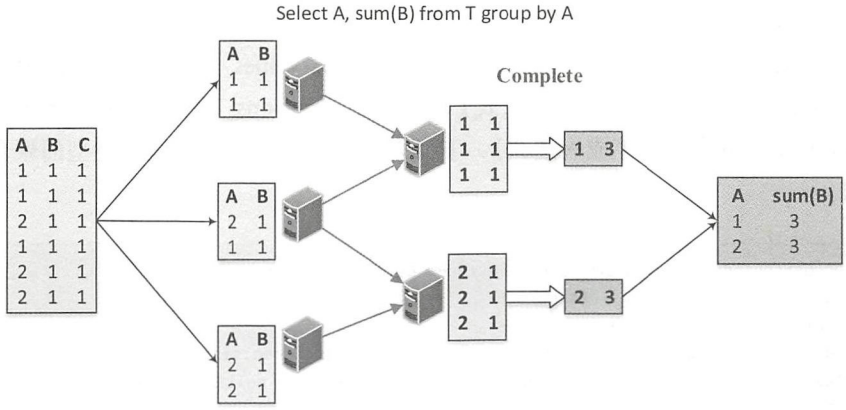


图 7.9 Complete 聚合模式

相比 Partial、Final 和 Complete 模式，PartialMerge 模式的聚合函数主要是对聚合缓冲区进行合并，但此时仍然不是最终的结果。ParitialMerge 主要应用在 distinct 语句中，如图 7.10 所示。聚合语句针对同一张表进行 sum 和 count（distinct）查询，最终的执行过程包含了 4 步聚合操作。第 1 步按照（A, C）分组，对 sum 函数进行 Partial 模式聚合计算；第 2 步是 PartialMerge 模式，对上一步计算之后的聚合缓冲区进行合并，但此时仍然不是最终的结果；第 3 步分组的列发生变化，再一次进行 Partial 模式的 count 计算；第 4 步完成 Final 模式的最终计算。

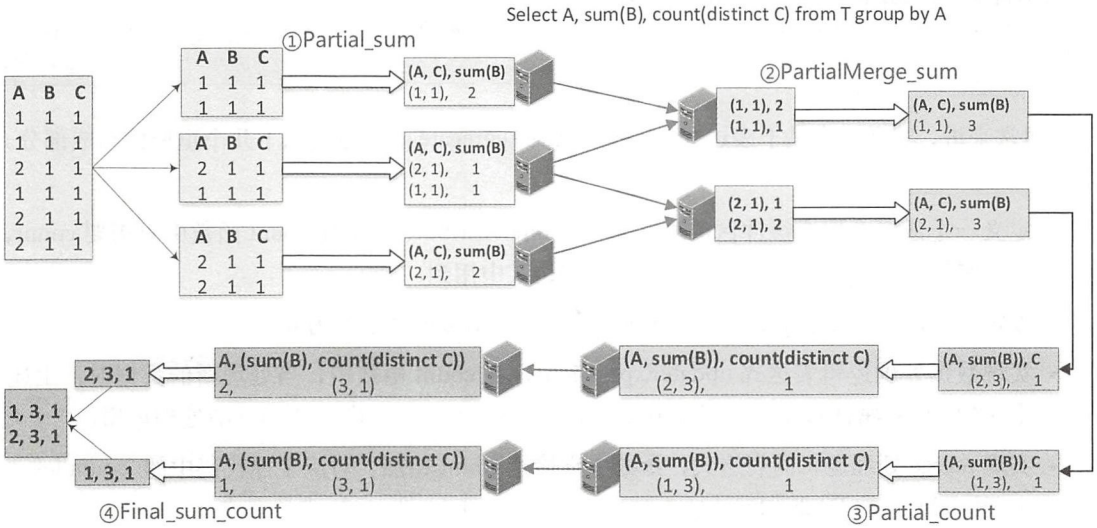


图 7.10 PartialMerge 聚合模式

7.2.2 DeclarativeAggregate 聚合函数

DeclarativeAggregate 聚合函数是一类直接由 Catalyst 中的表达式（Expressions）构建的聚合函数，主要逻辑通过调用 4 个表达式完成，分别是 initialValues（聚合缓冲区初始化表达式）、updateExpressions（聚合缓冲区更新表达式）、mergeExpressions（聚合缓冲区合并表达式）和 evaluateExpression（最终结果生成表达式）。下面以 Count 函数为例对这种类型的聚合函数的实现进行说明。

```
case class Count(children: Seq[Expression]) extends DeclarativeAggregate {
  override def nullable: Boolean = false
  override def dataType: DataType = LongType
  override def inputTypes: Seq[AbstractDataType] = Seq.fill(children.size)(AnyDataType)
  private lazy val count = AttributeReference( “count” , LongType, nullable = false)()
  override lazy val aggBufferAttributes = count :: Nil
  override lazy val initialValues = Seq(Literal(0L))
  override lazy val updateExpressions = {
    val nullableChildren = children.filter(_.nullable)
    if (nullableChildren.isEmpty) {
      Seq(count + 1L)
    } else {
      Seq(If(nullableChildren.map(IsNull).reduce(Or), count, count + 1L))
    }
  }
  override lazy val mergeExpressions = Seq(
    /* count = */ count.left + count.right
  )
  override lazy val evaluateExpression = count
  override def defaultResult: Option[Literal] = Option(Literal(0L))
}
```

通常来讲，实现一个基于表达式的 DeclarativeAggregate 函数包含以下几个重要的组成部分。

- 定义一个或多个聚合缓冲区的聚合属性 (bufferAttribute)，例如 count 函数中只需要 count，这些属性会在 updateExpressions 等各种表达式中用到。
- 设定 DeclarativeAggregate 函数的初始值，count 函数的初始值为 0。
- 实现数据处理逻辑表达式 updateExpressions，在 count 函数中，当处理新的数据时，上述定义的 count 属性转换为 Add 表达式，即 count+1L，注意其中对 Null 的处理逻辑。
- 实现 merge 处理逻辑的表达式，函数中直接把 count 相加，对应上述代码中的 “count.left + count.right”，由 DeclarativeAggregate 中定义的 RichAttribute 隐式类完成。
- 实现结果输出的表达式 evaluateExpression，返回 count 值。

7.2.3 ImperativeAggregate 聚合函数

不同于 DeclarativeAggregate 聚合函数基于 Catalyst 表达式的实现方式，ImperativeAggregate 聚合函数需要显式地实现 initialize、update 和 merge 方法来操作聚合缓冲区中的数据。一个比较显著的不同是，ImperativeAggregate 聚合函数所处理的聚合缓冲区本质上是基于行（InternalRow 类型）的。

聚合缓冲区是共享的，可能对应多个聚合函数，因此特定的 ImperativeAggregate 聚合函数会通过偏移量进行定位。例如，数据表有 3 列，分别是 key、x、y，查询语句中有两个求平均值的函数 avg(x) 和 avg(y)（注：假设这里用 ImperativeAggregate 的方式来实现平均值函数）。这两个函数共享聚合缓冲区 [sum1, count1, sum2, count2]，如图 7.11 所示，那么第一个 avg 函数的缓冲区偏移量为 0，第二个 avg 函数的缓冲区偏移量为 2，可以通过“mutableAggBufferOffset+fieldNumber”方式来访问具体的中间变量。

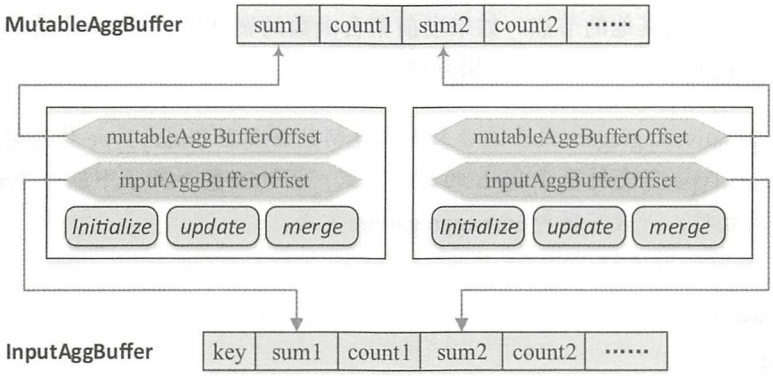


图 7.11 ImperativeAggregate 聚合函数

在 ImperativeAggregate 聚合函数中，还有输入聚合缓冲区（InputAggBuffer）的概念。InputAggBuffer 是不可变的，在聚合处理过程中将两个聚合缓冲区进行合并的时候，实现方式就是将该缓冲区的值更新到可变的聚合缓冲区中。除不可变外，InputAggBuffer 中相对聚合缓冲区还可能包含额外的属性，例如 group by 语句中的列，对应的缓冲区即 [key, sum1, count1, sum2, count2]。因此，在 ImperativeAggregate 聚合函数中还有 inputAggBuferOffset 的概念，用来访问 InputAggBuffer 中对应的中间值。

7.2.4 TypedImperativeAggregate 聚合函数

TypedImperativeAggregate[T] 聚合函数允许使用用户自定义的 Java 对象 T 作为内部的聚合缓冲区，因此这种类型的聚合函数是最灵活的。不过需要注意的是，TypedImperativeAggregate

聚合缓冲区容易带来与内存相关的问题。通常来讲，TypedImperativeAggregate 聚合函数的执行逻辑分为以下几步。

(1) 初始化聚合缓冲区对象。在聚合算子执行框架中，会调用 initialize 函数来创建一个空的聚合缓冲区。在 initialize 函数中，会执行 createAggregationBuffer 函数来获得初始化的缓冲区对象，并将其设置在全局的缓冲区中。

(2) 处理输入的数据行。如果当前聚合函数的聚合模式是 Partial 或 Complete，则执行框架会调用 update 函数来处理输入的数据行。在 update 函数的实现逻辑中，会从全局缓冲区获得缓冲对象，然后对其数据进行更新。如果当前聚合函数的聚合模式是 PartialMerge 或 Final，则执行框架会调用 merge 函数来处理数据，所处理的数据来自于其他节点序列化后的缓冲区对象。在 merge 函数的实现逻辑中，会将二进制数据反序列化成对应的 Java 对象，然后进行合并操作。

(3) 输出结果。如果当前聚合函数的聚合模式是 Partial 或 PartialMerge，则执行框架会调用 serializeAggregateBufferInPlace 函数将全局聚合缓冲区中的 Java 对象替换为序列化后的二进制数据，并将它们 Shuffle 到其他的节点。如果当前聚合函数的聚合模式是 Final 或 Complete，则执行框架会调用 eval 函数来计算最终的结果并将结果返回。

Note: TypedImperativeAggregate 聚合函数采用二进制数据类型 BinaryType 作为聚合缓冲区的存储方式，而 BinaryType 不支持基于 hash 的聚合方式，因此当聚合算子中包含 TypedImperativeAggregate 聚合函数时，一般实现为 sort aggregation 方式。

7.3 聚合执行

聚合执行本质上是将 RDD 的每个 Partition 中的数据进行处理。如图 7.12 所示，对于每个 Partition 中的输入数据即 Input（通过 InputIterator 进行读取），经过聚合执行计算之后，得到相应的结果数据即 Result（通过 AggregationIterator 来访问）。

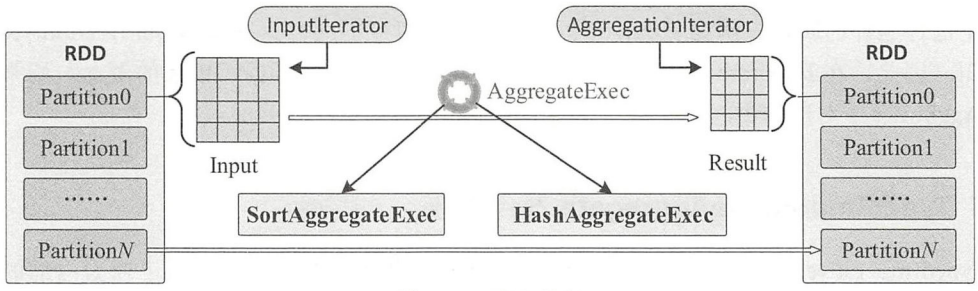


图 7.12 聚合执行

在 Spark 2.1 版本中，聚合查询的最终执行有两种方式：基于排序的聚合执行方式（SortAggregateExec）与基于 Hash 的聚合执行方式（HashAggregateExec）。在后续版本中，又加入了 ObjectHashAggregateExec 的执行方式（SPARK-17949），读者可自行研究其实现方式。常见的聚合查询语句通常采用 HashAggregate 方式，当存在以下几种情况时，会用 SortAggregate 方式来执行。

- 查询中存在不支持 Partial 方式的聚合函数：此时会调用 AggUtils 中的 planAggregateWithoutPartial 方法，直接生成 SortAggregateExec 聚合算子节点。
- 聚合函数结果不支持 Buffer 方式：如果结果类型不属于 [NullType, BooleanType, ByteType, ShortType, IntegerType, LongType, FloatType, DoubleType, DateType, TimestampType, DecimalType] 集合中的任意一种，则需要执行 SortAggregateExec 方式，例如 collect_set 和 collect_list 函数。
- 内存不足：如果在 HashAggregate 执行过程中，内存空间已满，那么聚合执行会切换到 SortAggregateExec 方式，这种情况将在 7.3.3 小节进行分析。

聚合查询的执行过程有一个通用的框架，主要接口定义在图 7.12 中的 AggregationIterator 中。本节后续内容首先介绍 AggregationIterator 的实现，然后分别分析 SortAggregate 与 HashAggregate 的技术实现原理。

7.3.1 执行框架 AggregationIterator

聚合执行框架指的是聚合过程中抽象出来的通用功能，包括聚合函数的初始化、聚合缓冲区更新合并函数和聚合结果生成函数等。这些功能都在聚合迭代器（AggregationIterator）中得到了实现。

如图 7.13 所示，聚合迭代器定义了 3 个重要的功能，分别是聚合函数初始化（initializeAggregateFunctions）、数据处理函数生成（generateProcessRow）和聚合结果输出函数生成（generateResultProjection）。SortBasedAggregationIterator 和 TungstenAggregationIterator 继承自 AggregationIterator，实现具体的操作，分别对应 SortAggregateExec 和 HashAggregateExec 两种执行方式。在 SortBasedAggregationIterator 和 TungstenAggregationIterator 中分别通过 processCurrentSortedGroup 与 processInputs 方法得到最终的聚合结果，而这两个方法均依赖上述 AggregationIterator 功能。

第一个功能聚合函数初始化可以细分为两个阶段，分别得到 funcWithBoundReference 和 funcWithUpdatedAggBufferOffset 表达式。第一阶段，funcWithBoundReference 执行 bindReference 或设置聚合缓冲区偏移量。针对 Partial 和 Complete 模式的 ImperativeAggregate 聚合函数，AttributeReference 表达式会转换为 BoundReference 表达式。例如，假设 Count(A) 处理的输入数

据行为整型的 (A, B, C)，经过转换后，得到的是 Count(BoundReference[0, Int, false])，提取出的是属性下标等信息；而对于 PartialMerge 和 Final 模式的 ImperativeAggregate 聚合函数，会设置输入缓冲区的偏移量（withNewInputAggBufferOffset）。第二阶段，funcWithUpdatedAggBufferOffset 设置 ImperativeAggregate 函数聚合缓冲区的偏移量（withNewMutableAggBufferOffset）。

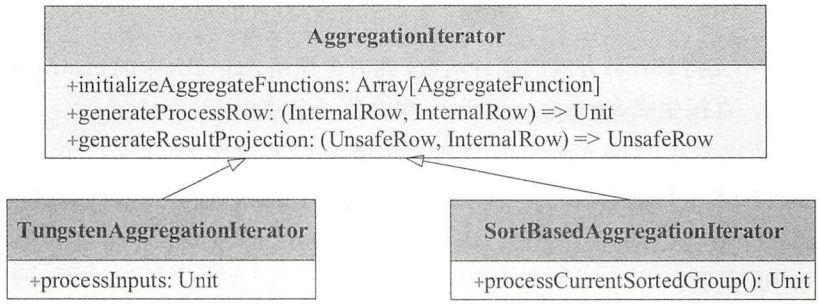


图 7.13 执行框架 AggregationIterator

第二个功能数据处理函数生成得到数据处理函数 processRow，其参数类型是 (InternalRow, InternalRow)，分别代表当前的聚合缓冲区 currentBufferRow 和输入数据行 row，输出是 Unit 类型。数据处理函数 processRow 的核心操作是获取各 Aggregation 中的 update 函数或 merge 函数。对于 Partial 和 Complete 模式，处理的是原始输入数据，因此采用的是 update 函数；而对于 Final 和 PartialMerge 模式，处理的是聚合缓冲区，因此采用的是 merge 函数。

第三个功能聚合结果输出函数生成计算最终的聚合结果，输入类型是 (UnsafeRow, InternalRow)，输出的是 UnsafeRow 数据行类型。对于 Partial 或 PartialMerge 模式的聚合函数，因为只是中间结果，所以需要保存 grouping 语句与 buffer 中所有的属性；对于 Final 和 Complete 聚合模式，直接对应 resultExpressions 表达式。特别注意，如果不包含任何聚合函数且只有分组操作，则直接创建 projection。

7.3.2 基于排序的聚合算子 SortAggregateExec

基于排序的聚合算子 SortAggregateExec 的实现比较简单。顾名思义，这是一种基于排序的聚合实现，在进行聚合之前，会根据 grouping key 进行分区并在分区内排序，将具有相同 grouping key 的记录分布在同一个 partition 内且前后相邻。如图 7.14 所示，聚合时只需要顺序遍历整个分区内的数据，即可得到聚合结果。

通过查看 SortAggregateExec 实现可知，requiredChildOrdering 中对输入数据的有序性做了约束，分组表达式列表（groupingExpressions）中的每个表达式 e 都必须满足升序排列，即 SortOrder(e, Ascending)，因此在 SortAggregateExec 节点之前通常都会添加一个 SortExec 节点。SortBasedAggregationIterator 是 SortAggregateExec 实现的关键，由于数据已经预先排好序，因此实

现相对简单，按照分组进行聚合即可。在其具体实现中，currentGroupingKey 和 nextGroupingKey 分别表示当前的分组表达式和下一个分组表达式，sortBasedAggregationBuffer 为其聚合缓冲区。比较重要的方法是 initialize 和 processCurrentSortedGroup，用来初始化基本信息和当前分组数据的处理。

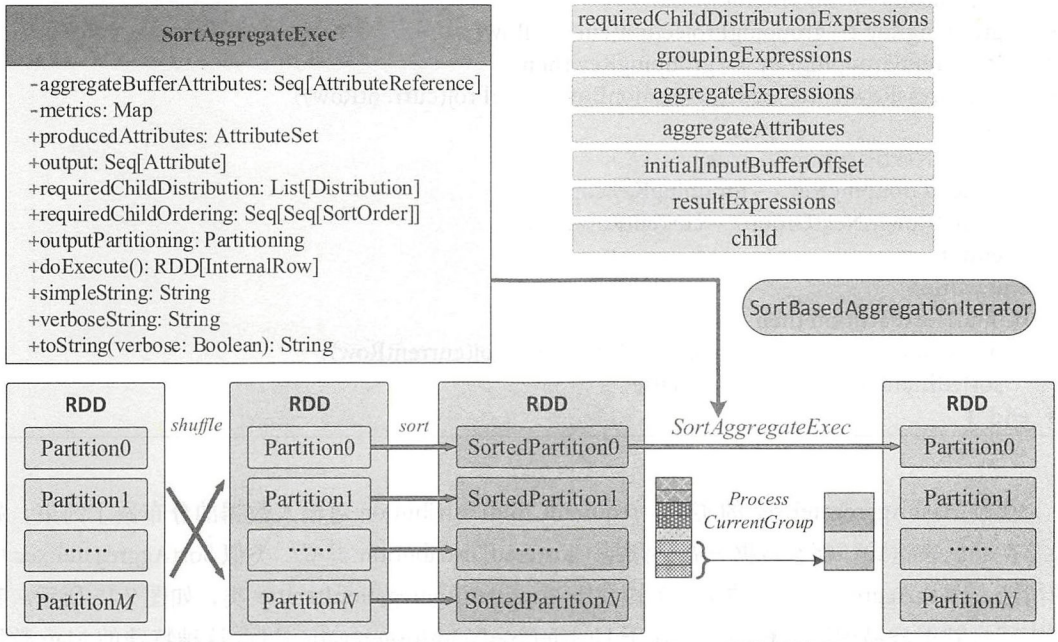


图 7.14 基于排序的聚合算子 SortAggregateExec 的执行过程

分组数据处理算法逻辑如 Algorithm 1 所示，由算法可知，在 while 循环过程中，不断获取输入数据并将其赋值给 currentRow，执行 groupingProjection 得到 groupingKey 分组表达式。如果当前的分组表达式 currentGroupingKey 和 groupingKey 相同，那么意味着当前输入数据仍然属于同一个分组内部，因此调用 AggregationIterator 中得到的 processRow 函数来处理当前数据。注意，数据处理之前首先通过 safeProj 将 currentRow 从 Unsafe 类型转换为 Safe 类型。

7.3.3 基于 Hash 的聚合算子 HashAggregateExec

HashAggregateExec 从逻辑上很好实现，只要构建一个 Map 类型的数据结构，以分组的属性作为 key，将数据保存到该 Map 中并进行聚合计算即可。然而，在实际系统中，无法确定性地申请到足够的空间来容纳所有数据，底层还涉及复杂的内存管理，因此相对 SortAggregateExec 的实现方式反而更加复杂。

Algorithm 1 SortBasedAggregationIterator.processCurrentSortedGroup

```
currentGroupingKey ← nextGroupingKey
findNextPartition ← false
processRow(sortBasedAggregationBuffer, safeProj(firstRowInNextGroup))
while !findNextPartition && inputIterator.hasNext do
    currentRow ← inputIterator.next()
    groupingKey ← groupingProjection(currentRow)
    if currentGroupingKey == groupingKey then
        processRow(sortBasedAggregationBuffer, safeProj(currentRow))
    else
        findNextPartition ← true
        nextGroupingKey ← groupingKey.copy()
        firstRowInNextGroup ← currentRow.copy()
    end if
end while
if !findNextPartition then
    processRow(sortBasedAggregationBuffer, safeProj(currentRow))
    sortedInputHasNewGroup ← false
end if
```

观察 HashAggregateExec 的实现，requiredChildDistribution 对输入数据的分布做了约束，如果存在分区表达式，那么数据分布必须是 ClusteredDistribution 类型。类似 SortAggregateExec 中的情形，HashAggregateExec 的实现关键在于 TungstenAggregationIterator 类，如图 7.15 所示。整体实现机制很容易理解，核心之处在于 UnsafeFixedWidthAggregationMap 这种特殊的 Map 数据结构。

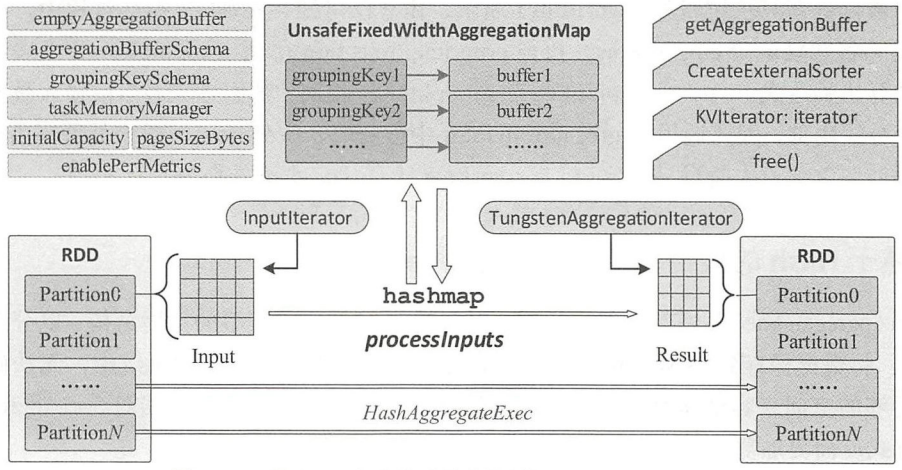


图 7.15　基于 Hash 的聚合执行过程 HashAggregateExec

实际上，HashAggregateExec 可能因为内存不足的原因退化为 SortAggregateExec 执行方式，因此 TungstenAggregationIterator 需要处理各种场景，相比 SortBasedAggregationIterator 的实现要复杂得多。从整体上来看，TungstenAggregationIterator 代码包含 8 部分内容。

- 聚合函数初始化相关的操作。
- 设置聚合缓冲区、数据处理和聚合结果生成的各种方法。
- 执行基于 Hash 聚合方式的各种方法。
- 切换到基于排序聚合的各种方法。
- 执行基于排序聚合方式的各种方法。
- 加载输入数据并处理输入数据的各种方法。
- 提供给外部调用的 Iterator 公共方法。
- 在不包含分组表达式或没有输入的情况下生成结果的辅助方法。

TungstenAggregationIterator 通过执行 processInputs 方法触发聚合操作，其主要逻辑如 Algorithm 2 所示。当分组表达式 groupingExpressions 为空时，过程比较简单，只要在获取输入数据的迭代过程中不断调用 processRow 函数来处理数据即可，因此算法省略了此部分内容。

这里以 processInputs 方法中的逻辑为主线梳理一下聚合操作的过程：在 while 循环中不断获取输入数据 newInput，然后得到分组表达式 groupingKey，并以此为 key 到 hashMap（注：UnsafeFixedWidthAggregationMap 类型，内部存储 groupingKey 与 UnsafeRow 的映射关系）中获取对应的聚合操作缓冲区（buffer），如果 buffer 不为空，则直接调用 processRow 处理。如果获取不到对应的 buffer，则意味着 hashMap 内存空间已满，这种情况下调用 destructAndCreateExternalSorter 方法将内存数据 spill 到磁盘以释放内存空间。注意，spill 到磁盘的数据会通过 UnsafeKVExternalSorter 数据结构访问，多次 spill 的外部数据还会进行合并操作。然后，再次从 hashMap 获取聚合缓冲区，此时如果无法获取，则会抛出 OOM 错误。

最后，检查全局 externalSorter 对象，如果不为空，则意味着聚合操作过程因为内存不足没能执行成功，部分数据存储在磁盘上。此时，将 hashMap 中最后的数据 spill 到磁盘并与 externalSorter 中的数据合并，然后调用 free 方法释放 hashMap，切换到基于排序的聚合执行方式。具体实现参考 switchToSortBasedAggregation 方法，其逻辑与 SortAggregateExec 方式的逻辑类似，这里不再展开讲解。

总体来看，HashAggregateExec 的执行过程还是比较清晰的，内存不足时借助磁盘空间实现聚合。中间涉及的一些特殊的数据结构（包括 UnsafeKVExternalSorter 和 UnsafeFixedWidthAggregationMap 等），其内部实现细节读者可以先暂时跳过，待熟悉了 Tungsten 相关的内容之后再自行研究。

Algorithm 2 TungstenAggregationIterator.processInputs

```
1: if !groupingExpressions.isEmpty then
2:   i ← 0
3:   while inputIter.hasNext do
4:     newInput ← inputIter.next()
5:     groupingKey ← groupingProjection.apply(newInput)
6:     buffer: UnsafeRow ← null
7:     if i < fallbackStartsAt._2 then
8:       buffer ← hashMap.getAggregationBufferFromUnsafeRow(groupingKey)
9:     end if
10:    if buffer == null then
11:      sorter ← hashMap.destructAndCreateExternalSorter()
12:      if externalSorter == null then
13:        externalSorter ← sorter
14:      else
15:        externalSorter.merge(sorter)
16:      end if
17:      i ← 0
18:      buffer ← hashMap.getAggregationBufferFromUnsafeRow(groupingKey)
19:      if buffer == null then
20:        throw new OutOfMemoryError
21:      end if
22:    end if
23:    processRow(buffer, newInput)
24:    i += 1
25:  end while
26:  if externalSorter != null then
27:    sorter ← hashMap.destructAndCreateExternalSorter()
28:    externalSorter.merge(sorter)
29:    hashMap.free()
30:    switchToSortBasedAggregation()
31:  end if
32: end if
```

7.4 窗口（Window）函数

窗口函数（Window Function）首先在 SQL-2003 标准中被提出，在 SQL-2008 标准中被细化，并在后续标准更新中多次扩展，最新的版本是 SQL-2011 标准。从功能本质来看，窗口函数和普通聚合函数（Aggregate Function）类似，都是对多行数据信息进行整合。不同之处在于，窗口函数中多了一个灵活的“窗口”，支持用户指定更加复杂的聚合行为（如数据划分和范围设置

等），因此在某些特殊分析场景下扮演着重要的角色。

7.4.1 窗口函数定义与简介

通常情况下，聚合操作会按照 Group By 子句对数据进行分组，然后在每个分组内执行聚合函数，得到一条结果。然而，这种常规的方式在面对一些复杂的分析需求时会显得捉襟见肘。例如，需要统计每个班前 5 名学生的成绩，或者需要计算每个学生的成绩与班级最高分的差距等。针对这类特殊的场景，窗口函数就有了用武之地。

窗口函数是 SQL 中的一类特别的函数，针对的是一个由 OVER 子句定义的“窗口”，即窗口函数的作用域。窗口是标准的 SQL 术语，形象地刻画了函数在计算过程中不断变化的数据范围。窗口函数可以通过计算每行周围窗口上的集合值来分析数据，例如，计算一定记录范围内、一定值域内或一段时间内的累计和移动平均值等。

具体来说，窗口和窗口函数在 Spark SQL 中的文法定义如下。窗口函数相比普通函数只不过多了 OVER 子句，其中的窗口信息（windowSpec）可以事先定义并在 SQL 中引用（windowRef 标签），也可以直接指定（windowDef 标签）。在 windowDef 标签的文法中，包括两个分支，分别对应 CLUSTER BY 和 PARTITION/DISTRIBUTE BY 开头的关键字。根据在实际系统中的观察，PARTITION BY 配合 ORDER BY 关键字的使用频率最高，因此这里以此作为分析的对象，其他情况读者可自行分析。

```
primaryExpression:  qualifiedName '(' (setQuantifier? expression (',' expression)*)? ')' (OVER
     windowSpec)? #functionCall

windowSpec : name=identifier  #windowRef
          | '('
            ( CLUSTER BY partition+=expression (',' partition+=expression)*
            | ((PARTITION | DISTRIBUTE) BY partition+=expression (',' partition+=expression)*)?
              ((ORDER | SORT) BY sortItem (',' sortItem)*)?)
            windowFrame?
           ')'              #windowDef

windowFrame : frameType=RANGE start=frameBound
            | frameType=ROWS start=frameBound
            | frameType=RANGE BETWEEN start=frameBound AND end=frameBound
            | frameType=ROWS BETWEEN start=frameBound AND end=frameBound

frameBound : UNBOUNDED boundType=(PRECEDING | FOLLOWING)
           | boundType=CURRENT ROW
           | expression boundType=(PRECEDING | FOLLOWING)
```

进一步来看，窗口函数涉及了 3 个核心元素，分别是分区（(PARTITION | DISTRIBUTE) BY）信息、排序（(ORDER | SORT) BY）信息和窗框定义（windowFrame）。

- 分区信息：分区元素由 PARTITION BY 子句定义，并被所有的窗口函数支持。类似 SparkPlan 中的 Partitioning，数据基于分区表达式执行 Hash 类型的 Shuffle 操作。在极端情况下，如果没有设定分区表达式，则所有数据都会集中到一个节点上。从另一个角度来看，分区可以算是对窗口的初步限制，只有值相同的数据才能进入同一个窗口。例如，窗口函数中使用 PARTITION BY ID，当前数据行的 ID 为 1，那么当前行所在的窗口中必然只能包括 ID 值为 1 的数据。
- 排序信息：排序元素定义分区内数据的顺序，在标准 SQL 中，所有函数都支持排序元素。排序子句所起的作用比较好理解，例如，对于排名函数（Rank），当使用降序排序时，排名函数返回对应分区内大于当前值的记录的个数加 1；当使用升序排序时，排名函数返回小于当前值的记录的个数加 1。实际上，某些窗口函数已经隐含地对数据有序性进行了要求，即使 SQL 语句中没有显示地指定，Spark SQL 后续解析时也会相应地添加。
- 窗框定义：本质上，窗框是一个在分区内对行进行进一步限制的筛选器，适用于聚合窗口函数，也适用于 3 个偏移函数，即 FIRST_VALUE、LAST_VALUE 和 NTH_VALUE。可以把这个窗口元素想象成基于特定的顺序、在当前行所在分区中定义的两个“点”，两点范围内的数据行才会参与计算。在标准的窗框描述中，可以用 ROWS 或 RANGE 关键字来定义如何选取开始行和结束行。ROWS 允许用相对于当前行的偏移行数来指定窗框的起点和终点；RANGE 则更具灵活性，可以以窗框起点和终点的值与当前行的值的差异来定义偏移行数。因此，ROWS 在物理级别（数据读取的数目）定义了窗口里有多少行，RANGE 则限定了排序之后的值在窗口里有多少行。此外，文法中的 PRECEDING 关键字可以定义窗口的上限，窗口从当前行向前若干行处开始，UNBOUNDED PRECEDING 表示没有上限（从第一行数据开始）。FOLLOWING 关键字定义窗口的下限，窗口从当前行向后若干行处结束，UNBOUNDED FOLLOWING 代表窗口没有下限（一直到最后一行数据）。表 7.2 列举了 3 个窗框的使用案例。

表 7.2　WindowFrame 案例

WindowFrame	范围
ROWS BETWEEN CURRENT ROW AND CURRENT ROW	当前行
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	开头到当前行
ROWS BETWEEN CURRENT AND UNBOUNDED FOLLOWING	当前行到结尾

下面以 row_number() 函数为例讲解窗口函数的使用。假设有关系表 exam（gradeID, classID, studentID, score），这 4 列分别代表年级 ID、班级 ID、学生 ID 和学生成绩，需要对每个年级每个班的学生按成绩排序并得到其排序号。那么，使用窗口函数 row_number() 的 SQL 语句及其执行过程如图 7.16 所示。

```
select studentID, row_number() over (partition by gradeID, classID order by score desc) as ranking from exam
```

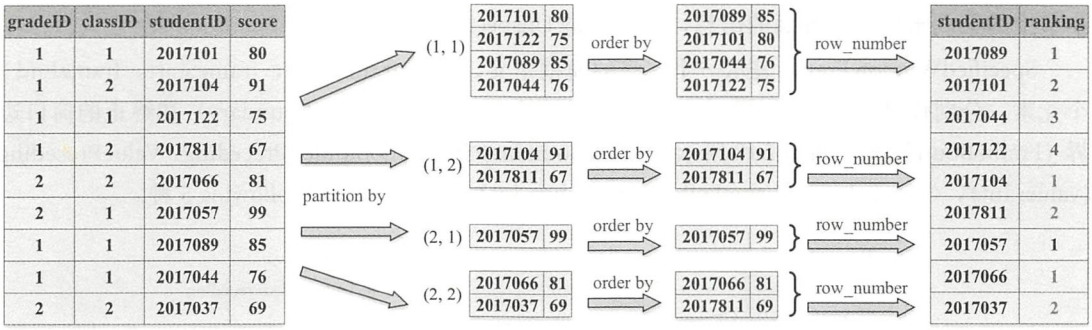


图 7.16 窗口函数实例

总体来看，窗口函数除输入、输出行相等外，还包括如下特性和优势：类似 Group By 的聚合，支持非顺序的数据访问；可以对窗口函数使用分析函数、聚合函数和排名函数；简化了 SQL 代码（消除 Join）并可以避免中间表。

7.4.2 窗口函数相关表达式

在分析窗口函数的执行过程之前，有必要对 Catalyst 中窗口函数的关键元素进行介绍，这部分内容定义在窗口函数相关的表达式中。

在 Catalyst 中，窗口表达式（WindowExpression）包含了窗口函数（WindowFunction）和窗框的定义。如图 7.17 所示，WindowExpression 的窗口函数（WindowFunction）是 Expression 类型，即上述案例中的 row_number() 函数；窗口定义是 WindowSpecDefinition 类型，代表 SQL 语句中 over 关键字之后括号里边的内容。

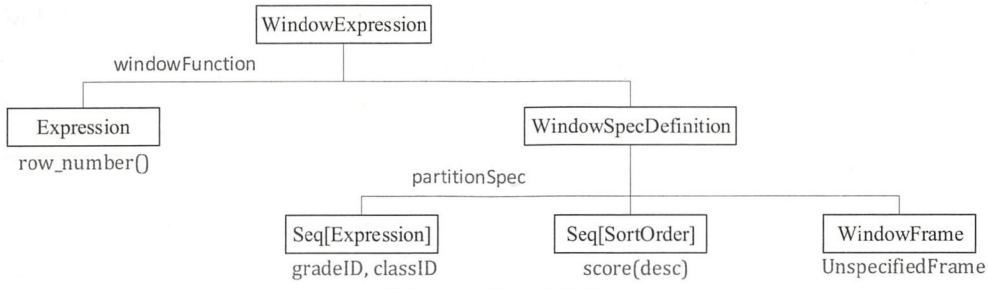


图 7.17 窗口表达式

在 WindowSpecDefinition 中包含了 7.4.1 小节中提到的窗口函数的 3 个核心元素：分区信息、排序信息和窗框定义。分区信息类型为 Seq[Expression]，在上述案例中表示按照 gradeID

和 classID 这两列进行分区；排序信息类型为 Seq[SortOrder]，对应 score 列的降序；窗框（WindowFrame）定义比较重要，有 UnspecifiedFrame 和 SpecifiedWindowFrame 两个子类，7.4.1 小节中的案例未指定这方面的信息，对应的是 UnspecifiedFrame 子类。

SpecifiedWindowFrame 表示一个完整的窗框定义，包含 frameType、frameStart、frameEnd 3 个元素，分别代表窗框类型（FrameType）、起始的窗口边界（FrameBoundary）和终止的窗口边界（FrameBoundary）。如图 7.18 所示，FrameBoundary 包含 UnboundedPreceding、ValuePreceding (value: Int)、CurrentRow、ValueFollowing（value: Int）和 UnboundedFollowing 5 种。

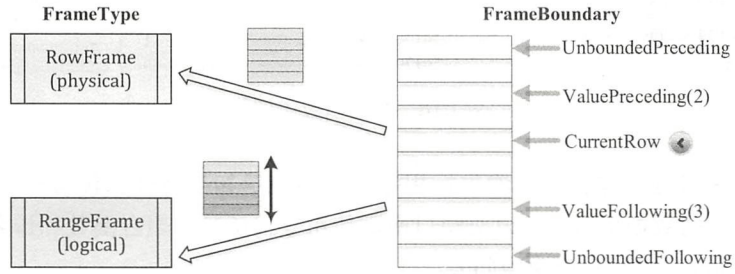


图 7.18　窗框 (WindowFrame) 的相关概念

FrameType 有 RowFrame 和 RangeFrame 两种。RowFrame 针对窗口分区中的所有数据行，当 ValuePreceding 和 ValueFollowing 作为窗口边界时，其中的 value 值代表物理偏移量，例如“ROW BETWEEN 1 PRECEDING AND 2 FOLLOWING”表示 4 行数据构成的窗框，即从当前行的前一行到后两行。RangeFrame 针对的是用于排序的列，当 ValuePreceding 和 ValueFollowing 作为窗口边界时，其中的 value 值代表逻辑偏移量，例如假设当前行的 score 值为 87，而窗框的边界定义为“RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING”，那么所对应的数据窗口为 score 在 [86, 88] 范围内的数据行。

窗口函数是 Expression 的子类，需要定义 WindowFrame 来设定该函数执行的默认窗口范围。目前在 Spark SQL 中，内置实现的窗口函数共有 8 个，如表 7.3 所示。

表 7.3　WindowFunction 种类

WindowFunction	用途
cume_dist	小于或等于当前值的行数占分组内总行数的比例
rank	生成的数据项在分组中的排名，排名相等会在名次中留下空位
dense_rank	生成的数据项在分组中的排名
percent_rank	(分组内当前行的 RANK 值 −1) 占 (分组内总行数 −1) 的比例
lead	统计窗口内往下第 k 行的值
lag	统计窗口内往上第 k 行的值
ntile	将分组数据按照顺序切分成 n 片，返回当前切片值
row_number	从 1 开始，按照顺序生成分组内记录的序列

在窗口函数中，lead 和 lag 都属于 OffsetWindowFunction 的子类，用于计算与偏移量相关的数据；其他都属于 AggregateWindowFunction 类型，在窗框内执行聚合计算。需要注意的是，在窗口查询中，除上述窗口函数外，也支持常见的函数。

7.4.3 窗口函数的逻辑计划阶段与物理计划阶段

在最终执行之前，有必要对包含窗口函数的查询中间过程进行分析。本节仍以 7.4.1 小节的案例作为分析对象，详细阐述其逻辑计划和物理计划间的转换。

上述 SQL 语句生成的完整的抽象语法树（AST）和之前的简单查询语法树大同小异，这里不再全部罗列，仅重点展示窗口函数的部分。如图 7.19 所示，Window 函数的主要特点是 FunctionCallContext 节点下面多了一些与窗口相关的节点信息。图 7.19 中的 WindowDefContext 节点代表窗口的定义，对于上述案例来讲，其中包含了一个 ExpressionContext 节点的列表（partition）来对应分区表达式，以及一个返回 SortItemContext 节点列表的函数（sortItem）来对应排序表达式。

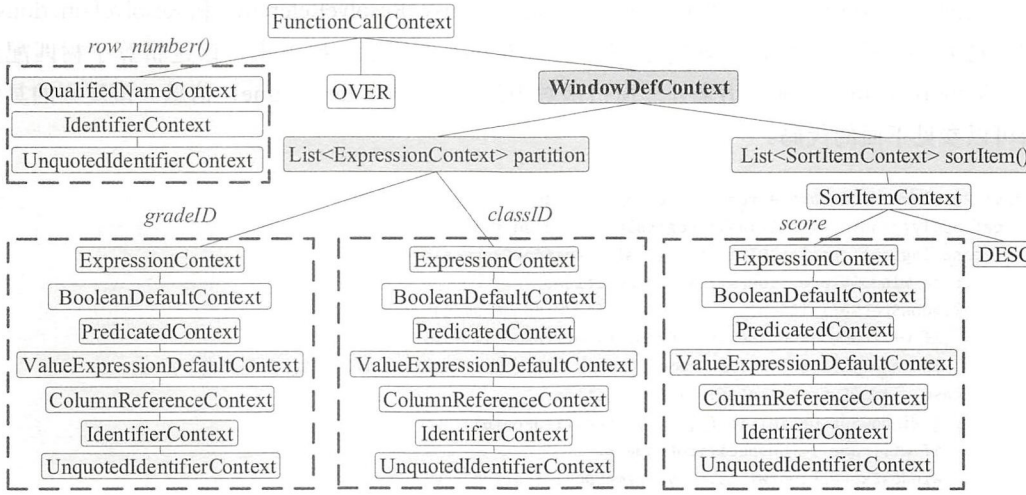


图 7.19 窗口函数 AST

图 7.19 中同时标出了函数名 row_number 和各个列所对应的子树。值得一提的是，如果 SQL 查询中涉及了窗框的相关信息，则 WindowDefContext 节点下面还会包含对应的 WindowFrame-Context 节点。

当 FunctionCallContext 节点下面出现 WindowDefContext 节点时，ASTbuilder 会将该函数对应的表达式封装成 WindowExpression 表达式。如图 7.20 所示为该查询生成的 Unresolved LogicalPlan 结构，有 UnresolvedRelation 和 Project 两个节点。

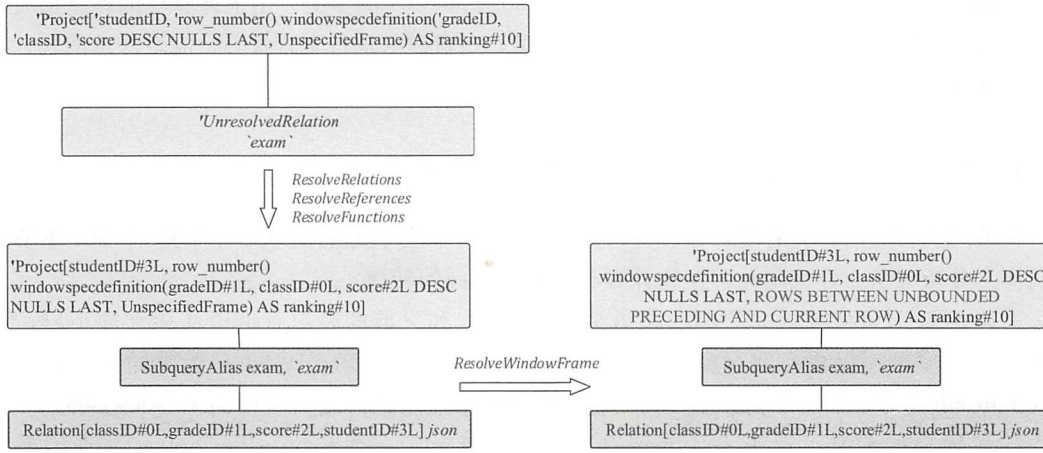


图 7.20　逻辑算子树（ResolveWindowFrame）

生成的 Unresolved LogicalPlan 经过 ResolveRelations、ResolveReference 和 ResolveFunctions 3 个解析规则的转换，得到的逻辑算子树如图 7.20 左下方所示。接下来，该逻辑算子树匹配到 ResolveWindowFrame 规则，用来处理窗口函数中的窗框（WindowFrame）信息。该规则的具体逻辑可以参见下面的代码。

```
object ResolveWindowFrame extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
      case logical: LogicalPlan => logical transformExpressions {
        case WindowExpression(wf: WindowFunction,
        WindowSpecDefinition(_, _, f: SpecifiedWindowFrame))
          if wf.frame != UnspecifiedFrame && wf.frame != f =>
          failAnalysis(s "Window Frame $f must match the required frame ${wf.frame}" )
        case WindowExpression(wf: WindowFunction,
        s @ WindowSpecDefinition(_, o, UnspecifiedFrame))
          if wf.frame != UnspecifiedFrame =>
          WindowExpression(wf, s.copy(frameSpecification = wf.frame))
        case we @ WindowExpression(e, s @ WindowSpecDefinition(_, o, UnspecifiedFrame))
          if e.resolved =>
          val frame = SpecifiedWindowFrame.defaultWindowFrame(o.nonEmpty, acceptWindowFrame = true)
          we.copy(windowSpec = s.copy(frameSpecification = frame))
      }
    }
}
```

ResolveWindowFrame 规则在解析并处理 WindowFrame 信息时有 3 种情况。

- 如果 WindowSpecDefinition 中指定了 WindowFrame(对应包含 SpecifiedWindow Frame 表达

式），而窗口函数中也设置了 WindowFrame 且与该 WindowFrame 不相同，则 SQL 语句抛出分析异常。

- 如果查询中未指定 WindowFrame，则将 WindowExpression 中的 WindowFrame 设置为窗口函数中的 WindowFrame 表达式。
- 查询中未指定 WindowFrame，而且函数不是 WindowFunction 类型，因此不包含 WindowFrame 信息，此时会将 WindowExpression 设置为默认的 WindowFrame 表达式。

查询实例中的情况对应第二种模式，WindowSpecDefinition 表达式中未定义 WindowFrame 信息（UnspecifiedFrame），因此直接使用窗口函数 row_number 中的 WindowFrame 设置（处理数据的范围是从开头到当前行）。生成的逻辑算子树如图 7.20 右下方所示，可以看到其中 WindowExpression 表达式发生了变化。

从前面的分析可知，在最初生成的逻辑算子树中，与窗口相关的内容都是以表达式来表示的，而不是单独的窗口节点。例如，在案例中窗口表达式依赖 Project 节点。而要执行窗口函数相关的查询，生成窗口机制相关的逻辑算子节点 Window 乃至后续的物理算子节点 WindowExec 的步骤是必不可少的。

实际上，这一步复杂的转换过程由逻辑计划阶段的 ExtractWindowExpressions 规则完成。顾名思义，ExtractWindowExpressions 规则用来从 WindowExpression 表达式中提取相关信息进行整合并生成单独的 Window 逻辑算子树节点。具体来讲，该规则处理以下 3 种情况。

- Project 节点的 projectList 表达式列表中包含的 WindowExpression 表达式。
- Aggregate 节点的 aggregateExpressions 表达式列表中包含的 WindowExpression 表达式。
- [Filter->Aggregate] 逻辑算子树结构模式。

针对每种情形，ExtractWindowExpressions 规则涉及以下两个重要的步骤。

(1) 表达式列表拆分：对于一个 NamedExpression 表达式列表（projectList 或 aggregateExpressions），将其分为两部分，其中一个是常规的表达式列表，另一个是所有的 WindowExpression 表达式列表。例如，假设 select 语句中的表达式如下：

```
col1, sum(col2 + col3) OVER (PARTITION BY col4 ORDER BY col5)。
```

该规则会提取“col1”“col2 + col3”“col4”“col5”，并根据列信息进行替换，那么该表达式列表会拆分为 [col1, col2 + col3 as _w0, col4 as _w1, col5 as _w2] 和 [sum(_w0) OVER (PARTITION BY _w1 ORDER BY _w2)] 两个表达式列表。

(2) Window 逻辑算子树节点创建：首先，对于上一步提取出来的所有 WindowExpression 表达式，根据其不同的窗口定义 (WindowSpecDefinition) 进行分组（注：相同的 WindowSpecDefinition

对应的窗口函数可以放在一起进行处理）；接着，针对每个不同的窗口定义（WindowSpecDefinition），创建 Window 逻辑算子树节点并将其插入到逻辑算子树中，需要注意的是每个 Window 逻辑算子节点相应地处理一个 WindowSpecDefinition 的窗口函数。

经过 ExtractWindowExpressions 规则的处理，得到的逻辑算子树如图 7.21 所示。可以看到，WindowExpression 已经被提取出来生成了 Window 逻辑算子节点，同时围绕该节点添加了 3 个 Project 节点。

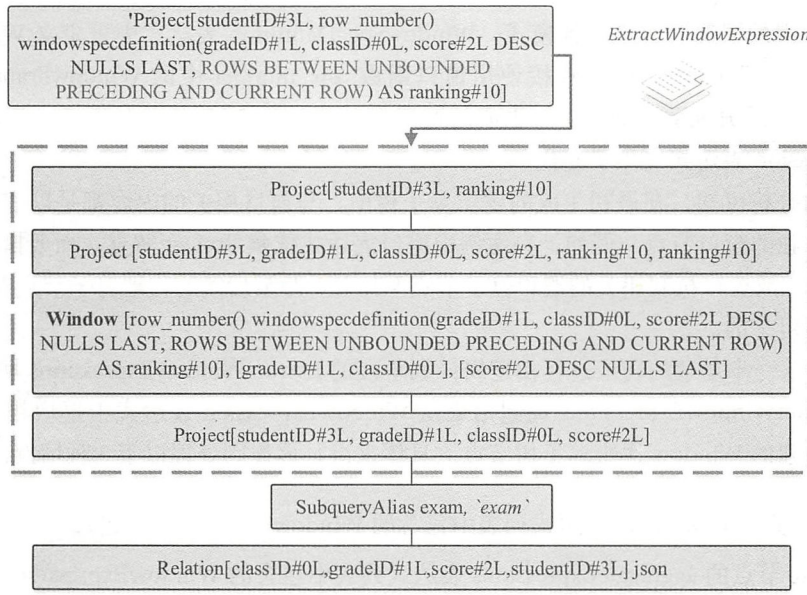


图 7.21　逻辑算子树 (ExtractWindowExpressions)

实际上，在 Analyzer 中，对于与 Window 相关的逻辑算子树，除上述两条规则外，还有 ResolveWindowOrder 规则，主要用来对排序功能进行验证，确保窗口语句中包含排序语句或 Rank 之类的窗口函数。因为本案例中已经包含了排序语句，所以上述过程没有体现出来。

在图 7.21 所示的 Analyzed LogicalPlan 的基础上，Optimizer 会进行进一步的优化。经过别名消除和 Project 节点整合之后，得到的逻辑算子树如图 7.22 所示，有 Relation、Project、Window 和 Project 4 个逻辑算子节点。

从逻辑算子树生成物理算子树的过程比较简单，分别应用 BasicOperatiors 中的映射策略和 FileSourceStrategy 策略，生成对应的 SparkPlan，如图 7.23 所示。在物理算子树中，WindowExec 节点对应 Window 逻辑算子节点，很显然，该节点在 Partitioning 和 Ordering 方面都有正确性的需求，因此在最后阶段的 EnsureRequirements 规则中，会添加 Exchange 节点进行 Shuffle 操作，以及添加 SortExec 节点进行排序操作，得到包含 6 个节点的物理执行计划。

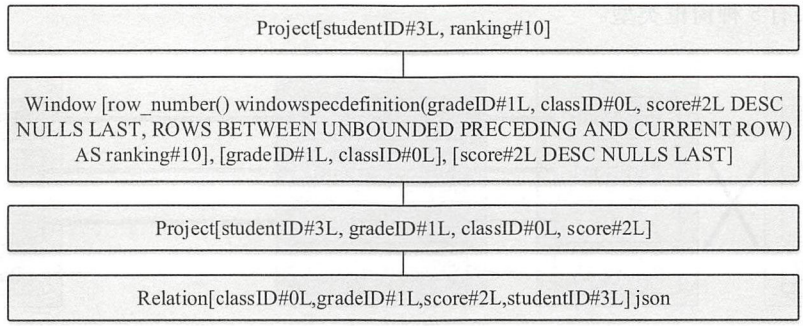


图 7.22 最终逻辑算子树

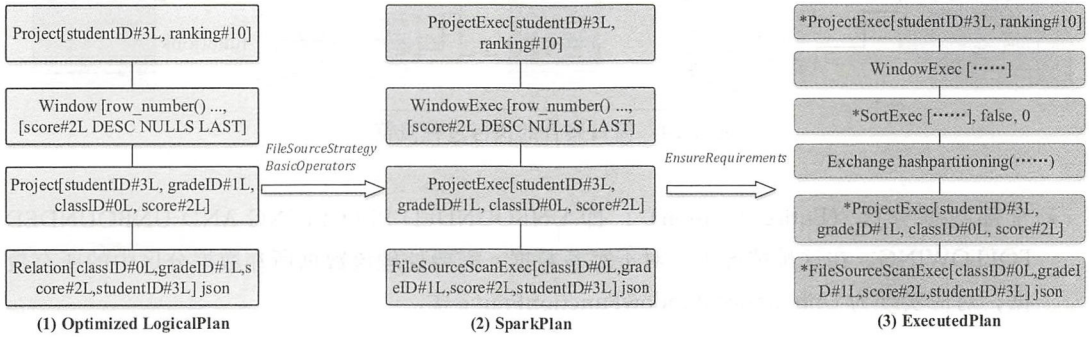


图 7.23 物理执行计划

7.4.4 窗口函数的执行

生成物理执行计划之后，就提交到 Spark 集群进行计算了。窗口函数的执行逻辑在 WindowExec 中实现，本小节将对其进行详细分析。

窗口聚合的执行过程如图 7.24 所示，WindowExec 类的 requiredChildDistribution 和 requiredChildOrdering 方法分别规定了输入数据分布和有序性要求，因此在执行 WindowExec 之前，Exchange 和 Sort 完成重分区及分区内数据的排序。对应于本节的查询实例，Exchange 会按照 (gradeID, classID) 进行分区，并按照 score 进行排序。

在 WindowExec 执行中，有两个比较重要的概念：一个是 FramedFunctions，记录不同的窗口表达式（WindowExpression）间的映射关系，同一个窗口可能包含多个窗口表达式，也就是多个窗口函数；另一个是 AggregateProcessor，类似聚合语句中的 AggregateIterator，通过执行具体的窗口函数进行实际的计算。

物理执行计划 WindowExec 用于在单个有序的数据分区中计算并输出窗口聚合结果，与普通聚合过程不同的是，窗口聚合会根据窗口函数的窗框等设置对每一行数据进行计算。根据前

面的分析，共有 5 种窗框类型。

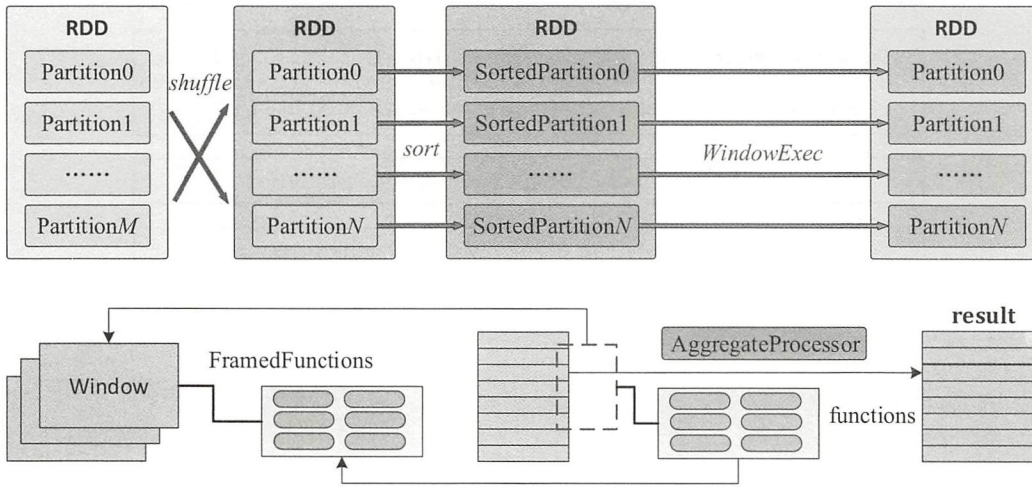


图 7.24　窗口聚合的执行过程概览

- 全部数据分区（Entire Partition）：即 UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING。在这种情况下，对于每条数据，需要处理该数据所在数据分区中的所有数据，对应实现为 UnboundedWindowFunctionFrame 类。
- 扩张框（Growing Frame）：即 UNBOUNDED PRECEDING AND ..... 在这种情况下，每次都会移动到新的数据行进行处理，并添加一些数据行扩展该框架，在这种类型的窗框中数据不会被移除，只会不停地加入新数据，窗口范围不停地“扩张”，对应的实现为 UnboundedPrecedingWindowFunctionFrame 类，案例中的 row_number() 函数就属于这种类型。
- 收缩框（Shrinking Frame）：该框架只会移除数据，即..... AND UNBOUNDED FOLLOWING。在这种情况下，每次都会移动到新的数据行进行处理，并从窗框中移除一些数据行，这种类型的窗框中不会添加数据，窗口不停地“收缩”，对应的实现为 UnboundedFollowing-WindowFunctionFrame 类。
- 移动框（Moving Frame）：每次处理到新的数据行，都会添加一些数据，同时也会删除一些数据，例如（1 PRECEDING AND CURRENT ROW）或（1 FOLLOWING AND 2 FOLLOWING），对应的实现为 SlidingWindowFunctionFrame 类。
- 偏移框（Offset Frame）：该窗框仅包含一行数据，即距离当前数据行特定偏移量的数据。这里需要注意的是，偏移框仅适用于 OffsetWindowFunction 类型的窗口函数。

这 5 种类型的窗框可以统一抽象为窗口函数执行框架，如图 7.25 所示。从抽象层面来看，

窗口函数执行阶段只处理两件事情，即准备数据行缓冲区（RowBuffer）和写入结果，对应的实现为 prepare 和 write 函数。

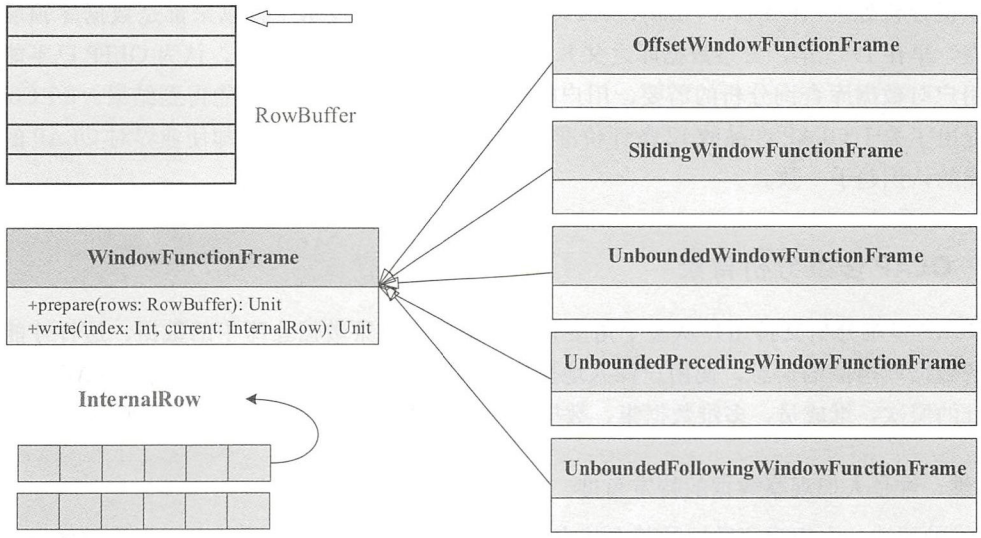


图 7.25 窗口函数执行框架

行缓冲区（RowBuffer）服务于单个窗口数据分区，实例中每个 (gradeID, classID) 分区的数据对应一个 RowBuffer。考虑到窗口函数处理过程中需要反复扫描数据行，因此 RowBuffer 在本质上起到物化（Materialize）分区数据行的作用。RowBuffer 定义为抽象类，支持 size、next、skip 和 copy 4 种操作，具体实现包括 ArrayRowBuffer 和 ExternalRowBuffer 两种，分别对应内存和外存的情况。

根据 WindowExec 类 doExecute 方法可知，整体的执行逻辑分为两步。

（1）按分区读取数据（fetchNextPartition），并将其保存在 ArrayBuffer 中，在此过程中先构造一个 ArrayRowBuffer 存储数据，如果超过阈值（默认为 4096），则切换为基于磁盘的 UnsafeExternalSorter 数据结构；当前分区数据读取结束后，根据 UnsafeExternalSorter 是否为空，判断分区数据保存在 ArrayRowBuffer 还是 ExternalRowBuffer 数据结构中。

（2）遍历 RowBuffer 中的数据，逐条执行 write 操作，最终调用 AggregateProcessor（注：封装 row_number() 函数）中的 update 等方法完成计算。

AggregateProcessor 中的实现方式和普通的聚合操作的实现方式类似，这里不再展开讲解。值得一提的是，RowBuffer 在后续版本中替换成了更加安全、高效的实现（ExternalAppendOnlyUnsafeRowArray），读者可进一步自行研究。

7.5 多维分析

联机分析处理（On-Line Analytical Processing，OLAP）技术一直以来都是数据库领域的研究热点。早在 1993 年，关系数据库之父 E.F.Codd 就提出了 OLAP 概念，认为 OLTP 已不能满足终端用户对数据库查询分析的需要，用户的决策分析需要大量计算才能得到结果。E.F.Codd 还同时提出了关于 OLAP 产品的 12 条评价准则，虽未得到一致公认，但却使业界对 OLAP 的概念和功能的认识趋于一致。

7.5.1 OLAP 多维分析背景

OLAP 多维分析支持用户从多个角度和多个侧面去考察数据仓库中的数据，这样才能深入地了解数据所蕴涵的信息，使用户深入地挖掘隐藏在数据背后的商业模式。在 OLAP 中，涉及维、维的层次、维成员、多维数据集、数据单元、多维数据集的度量值等基本概念。

- 维：维是人们观察数据的特定角度。
- 维的层次：人们观察数据的某个特定角度（某个维）还可以存在细节、程度等多个方面的描述。
- 维成员：维的一个取值称为该维的一个维成员。如果一个维是多层次的，那么该维的维成员是在不同维层次的取值组合。
- 多维数据集：多维数据集是决策支持的支柱，也称为立方体。
- 数据单元：多维数据集的取值称为数据单元。
- 多维数据集的度量值：在多维数据集中有一组度量值，这些值基于多维数据集中事实表的一列或多列。度量值是多维数据集的核心值，是用户在应用中最终需要查看的数据。

为了保证信息处理所需的数据以合适的粒度、合理的抽象程度和标准化程度存储，数据在物理上分为 3 种存储结构：基于多维数据库的 OLAP 存储结构（MOLAP）、基于关系数据库的 OLAP 存储结构（ROLAP）和混合型的 OLAP 存储结构（HOLAP）。

MOLAP（Multidimensional OLAP）：多维 OLAP 利用一种专有的多维数据库来存储 OLAP 分析所需要的数据，数据采用 n 维数组的多维方式存储，并以多维视图的方式显示。MOLAP 结构的主要优点是能迅速地响应分析人员的分析请求并快速地将分析结果返回给用户。缺点是限制了 MOLAP 结构的灵活性，主要表现在以下几个方面：首先，用户很难对维数进行动态改变，每增加一维都会使数据库的规模急剧增加，所需的预处理时间也会大大增加；其次，对数据变化的适应能力弱，当数据或计算频繁变化时，重复计算量非常大，甚至还需重新构建多维数据

库；最后，处理大量细节数据的能力不足，MOLAP 较强的预处理能力限制了处理大量细节数据的能力。

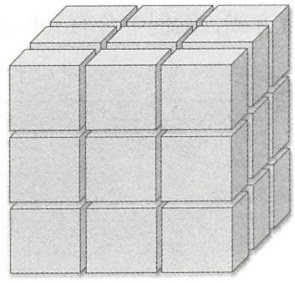
ROLAP（Rational OLAP）：关系型 OLAP 对传统数据库进行了扩充以实现对数据仓库的联机分析处理。ROLAP 的底层数据库是关系型数据库，而不是多维数据库。ROLAP 一般采用星型（Star Schema）或雪花模型（Snowflake Schema）表达多维数据视图。在 OLAP 结构中数据的预处理程度一般较低（注：预处理程度高，数据冗余量大，管理和维护更加复杂）。ROLAP 的主要特点是灵活性强，可以动态定义统计或计算方式。ROLAP 的缺点是对用户分析请求处理的时间比 MOLAP 长。

HOLAP（Hybrid OLAP）：MOLAP 与 ROLAP 存储结构迥然不同，有着各自的优缺点，基于优化的考虑，混合型 OLAP（HOLAP）被提出。HOLAP 结构不是 MOLAP 与 ROLAP 结构的简单组合，而是这两种结构技术优点的有机结合，能够满足用户各种复杂的分析请求。一个完整的 HOLAP 系统需要遵循以下几个准则。

- 维数能够被动态更新，一个真正的 HOLAP 不但可以提供对数据的实时存取，还可以根据不断变化的结构对维数进行更新。
- 可以根据关系数据库的元数据产生多维视图。
- 可以快速地存取各种级别的汇总数据。
- 能够适应大数据量数据的分析。
- 可以方便地对计算和汇总算法进行维护和修改。

7.5.2 Spark SQL 多维查询

相比传统数据仓库，Spark SQL 中对多维分析的支持较为简单，底层也并不提供专门的存储结构。本章开头已经介绍过，SQL 文法中多维分析的关键字有 cube、rollup 和 grouping sets 3 种，图 7.26 展示了 Spark SQL 中多维分析的使用案例。



```
Q1: select gradeID, classID, max(score) from exam
    group by gradeID, classID
    with cube
```

```
Q2: select gradeID, classID, max(score) from exam
    group by gradeID, classID
    with rollup
```

```
Q3: select gradeID, classID, max(score) from exam
    group by gradeID, classID
    grouping sets ((gradeID, classID), gradID)
```

图 7.26 多维分析使用案例

这里以 cube 关键字为例进行讲解，图 7.26 中的 Q1 在 group by 子句中指定了维度列（例如，案例中的 gradeID 和 classID）和关键字 with cube。查询结果包含维度列中各维度值的所有可能组合，以及与这些维度值组合匹配的基础行中的聚合值，所以 Q1 的效果等价于以下多条 SQL 聚合查询结果的组合。

```
select gradeID, classID, max(score) from exam group by gradeID, classID
union
select gradeID, null, max(score) from exam group by gradeID, null
union
select null, classID, max(score) from exam group by null, classID
union
select null, null, max(score) from exam group by null, null。
```

图 7.27 通过实际数据展示了含有 cube 运算符的 SQL 执行过程，整体思路比较好理解。根据 (gradeID, classID) 列是否为 null，共有 4 种情况，最终得到 9 条聚合结果。

```
select gradeID, classID, max(score) from exam group by gradeID, classID with cube
```

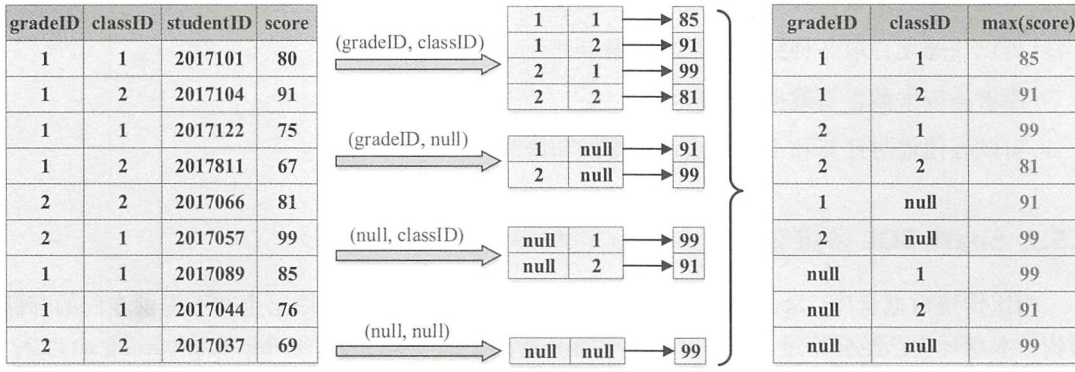


图 7.27　多维分析实例

此外，rollup 和 grouping sets 与 cube 的不同之处在于列的组合方式，案例中含有 rollup 关键字的 SQL 语句等价于：

```
select gradeID, classID, max(score) from exam group by gradeID, classID
union
select gradeID, null, max(score) from exam group by gradeID, null
union
select null, null, max(score) from exam group by null, null;
```

含有 grouping sets 关键字的 SQL 语句等价于：

```
select gradeID, null, max(score) from exam group by gradeID, null
union
select null, classID, max(score) from exam group by null, classID。
```

可以看到，多维分析在执行过程中一般都会产生空值（null）。在这种情况下，往往会带来一个问题：如何区分多维分析算子产生的 null 值和实际数据中的 null 值？常用的方法是使用 grouping 函数来区分，如果是由多维分析产生的 null 值，则 grouping 函数返回 1；如果是数据本身的 null 值，则 grouping 函数返回 0。

7.5.3 多维分析 LogicalPlan 阶段

根据前面的文法定义可知，多维分析语句（cube、rollup 和 grouping sets）定义在聚合语句中，因此这里对多维分析查询的原理研究直接从逻辑计划开始。先回顾一下聚合语句对应的抽象语法树，图 7.28 展示了 7.5.2 小节 cube 语句对应的树型结构。

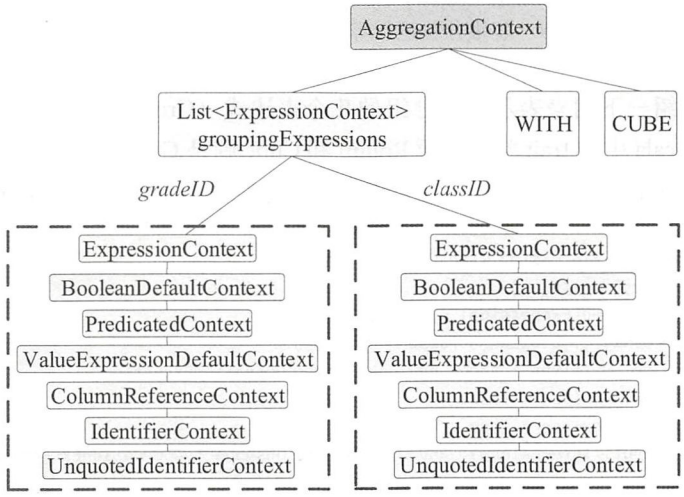


图 7.28 多维分析案例的语法树

可以看到，AggregationContext 节点包含了一系列 ExpressionContext 节点列表，分别对应 group by 语句中的各列信息（案例中的 gradeID 和 classID 列）。该语句中包含 cube 关键字，因此这里的 CUBE 子节点不为空。如果包含 rollup 关键字，则对应的是 ROLLUP 节点；同样，如果包含 grouping sets 关键字，则对应的是 GROUPING 节点和相关的分组表达式。

上述抽象语法树生成逻辑算子树的工作由 AstBuilder 完成。不同于一般的聚合查询，AstBuilder 对多维分析的处理多了一步针对 group by 语句的封装，具体逻辑可以参见其中的 withAggregation 函数实现。对于该案例，传递给 Aggregate 节点的 group by 表达式处理为 Seq(Cube(groupByExpressions))，生成的逻辑算子树如图 7.29 所示。

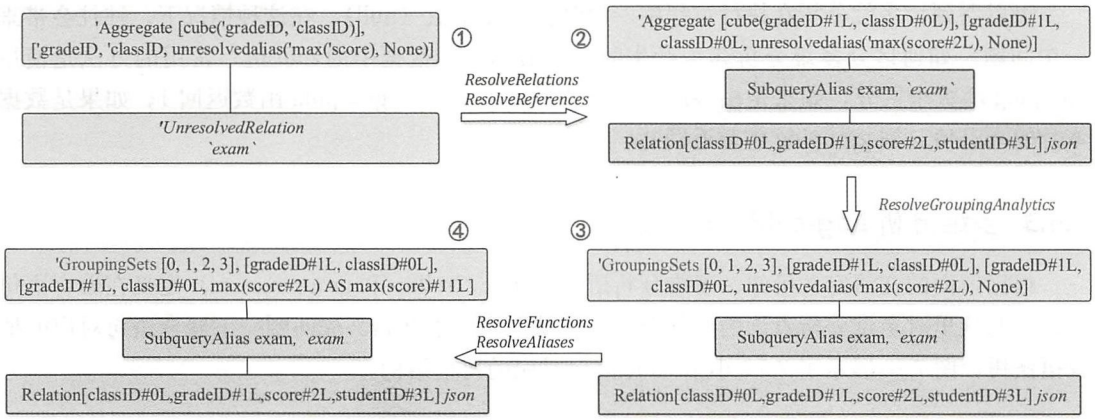


图 7.29　多维分析逻辑算子树（GroupingSets 生成）

这里有必要介绍一下用来表示多维分组的几个表达式（GroupingSet、Rollup 和 Cube），其中 GroupingSet 是 Scala 中的 trait 类型，而 Rollup 和 Cube 则是 GroupingSet 的继承。GroupingSet 的实现如以下代码所示。由此可知，其主要作用在于记录聚合语句中包含了哪些分组表达式。

```
trait GroupingSet extends Expression with CodegenFallback {
  def groupByExprs: Seq[Expression]
  override def children: Seq[Expression] = groupByExprs
  override lazy val resolved: Boolean = false
  override def dataType: DataType = throw new UnsupportedOperationException
  override def foldable: Boolean = false
  override def nullable: Boolean = true
  override def eval(input: InternalRow): Any = throw new UnsupportedOperationException
}
```

对抽象语法树其他部分的处理与通常的情况类似，不再赘述。经过 AstBuilder 处理之后，生成的 Unresolved LogicalPlan 包含了 UnresolvedRelation 和 Aggregate 两个节点，如图 7.29 所示。接下来，进入非常重要的 Analyzer 阶段。

在前面分析的过程中应用了常见的 ResolveRelations 和 ResolveReferences 规则，分别解析数据表信息和数据列信息。接下来，对生成的逻辑算子树起作用的是 ResolveGroupingAnalytics 规则，该规则可以算是多维分析中最重要的部分。在分析该规则的逻辑之前，先介绍一下会用到的 GroupingSets 逻辑算子树节点。代码如下所示，可以看到，GroupingSets 和 Aggregate 内部的

内容类似，都记录了 group by 分组表达式列表和 aggregation 聚合表达式列表；不同之处在于多了一个整型列表（bitmasks），列表中的每个数都用来定位 groupByExprs 列表中选中的表达式。例如，假设 bitmasks 列表为 (1, 2)，二进制表示分别为 (01, 10)，而 groupByExprs 列表为 (gradeID, classID)，那么 bitmasks 表示选中的 group by 列表分别为 (gradeID, null) 和 (null, classID)，其中二进制 0 表示对应下标的列参与 group by 操作，二进制 1 表示对应下标的列不参与 group by 操作 (这里表示为 null)。

```
case class GroupingSets(bitmasks: Seq[Int], groupByExprs: Seq[Expression],
    child: LogicalPlan, aggregations: Seq[NamedExpression]) extends UnaryNode {
  override def output: Seq[Attribute] = aggregations.map(_.toAttribute)
  override lazy val resolved: Boolean = false
}
```

总体来讲，ResolveGroupingAnalytics 规则进行以下 3 方面的处理或转换。

- 生成 GroupingSets 节点：针对 Aggregate 节点，如果其中 group by 语句对应的是 cube 或 rollup 表达式，计算相应的 bitmasks，将 Aggregate 节点转换为 GroupingSets 节点。需要注意的是，grouping sets 语句在 AstBuilder 中完成了相应的转换。
- 生成 Expand+Aggregate 节点：针对 GroupingSets 节点，生成 Expand 逻辑算子树节点加上 Aggregate 逻辑算子树节点。这方面的处理是 ResolveGroupingAnalytics 规则的核心，后面将会重点分析。
- 替换多维分析函数（grouping_id 和 grouping 函数）：将逻辑算子树某些节点中（主要存在 Filter 或 Sort 节点）的表达式所包含的多维分析函数，替换为对应生成的列名（在 Expand 节点中生成）。

在替换多维分析函数的过程中，由于还存在未解析（resolved=false）的表达式，所以这里的操作仅仅是将包含多维分析的 Aggregate 节点转换为 GroupingSets 节点。可以看到 cube 生成的 bitmasks 列表为 [0, 1, 2, 3]，正好对应 4 种情形。接下来，Analyzer 匹配 ResolveFunctions 和 ResolveAliases 规则，解析得到新的逻辑算子树。

然后，Analyzer 继续将 ResolveGroupingAnalytics 规则应用在新的逻辑算子树上，生成的结果如图 7.30 所示。由此可知，GroupingSets 节点转换为 Aggregate+Expand+Project 3 个节点的组合。顾名思义，Expand 表示“扩展”，多维分析在本质上相当于执行多种组合的 group by 操作，因此 Expand 所起的作用就是将一条数据扩展为特定形式的多条数据，例如本案例 cube 对应 4 条数据。Expand 中有两个重要的信息，一个是表示扩展数据的 projections 列表，另一个是表示输出的 output 表达式。

为了支持 Expand 节点的操作，在 Expand 节点之前一般还会添加一个 Project 节点，所做的处理比较简单，额外添加 group by 语句中经过别名处理后的列。基于该处理，在 Expand 中还会根据对应的 bitmasks 数字生成对应的 grouping_id，同时在 output 中也会添加 spark_grouping_id 这一列。从图 7.30 中也可以看到，Aggregate 节点中的 group by 表达式也多了 spark_grouping_id 列。

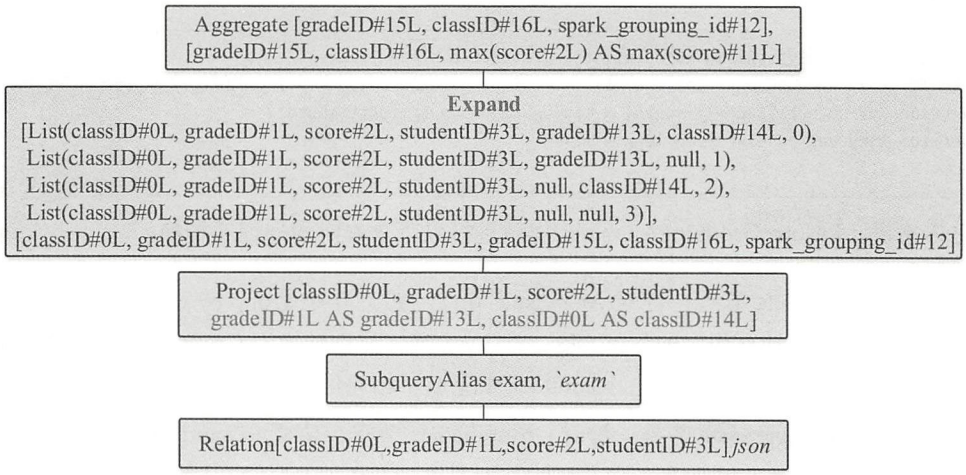


图 7.30　多维分析逻辑算子树（Expand 生成）

在 Analyzed LogicalPlan 生成后，如图 7.31 上部分所示，进入 Optimizer 优化阶段。对于上述逻辑算子树，在 Optimizer 中将会应用 EliminateSubqueryAliases（子查询别名去重）规则、ColumnPruning（列剪裁）规则、CollapseProject（投影整合）规则和 RemoveAliasOnlyProject 规则进行优化。每条规则的效果在此不再展开，最终生成的 Optimized LogicalPlan 如图 7.31 下部分所示。

总体来看，Spark SQL 多维分析实现的主要思路是：在基础数据上添加特殊的分组“标签”，这样每一条基础数据都会“扩展”成多条新的数据，然后针对新的数据执行一次聚合操作。在上述 Spark 2.1 版本的实现方式中，涉及大量的位操作，不利于理解和开发，后续版本中有了重构，读者可自行参考。

7.5.4 多维分析 PhysicalPlan 与执行

从逻辑算子树生成物理算子树的过程比较简单，分别应用 BasicOperatiors 中的映射策略和 FileSourceStrategy 策略，生成对应的 SparkPlan，如图 7.32 所示。

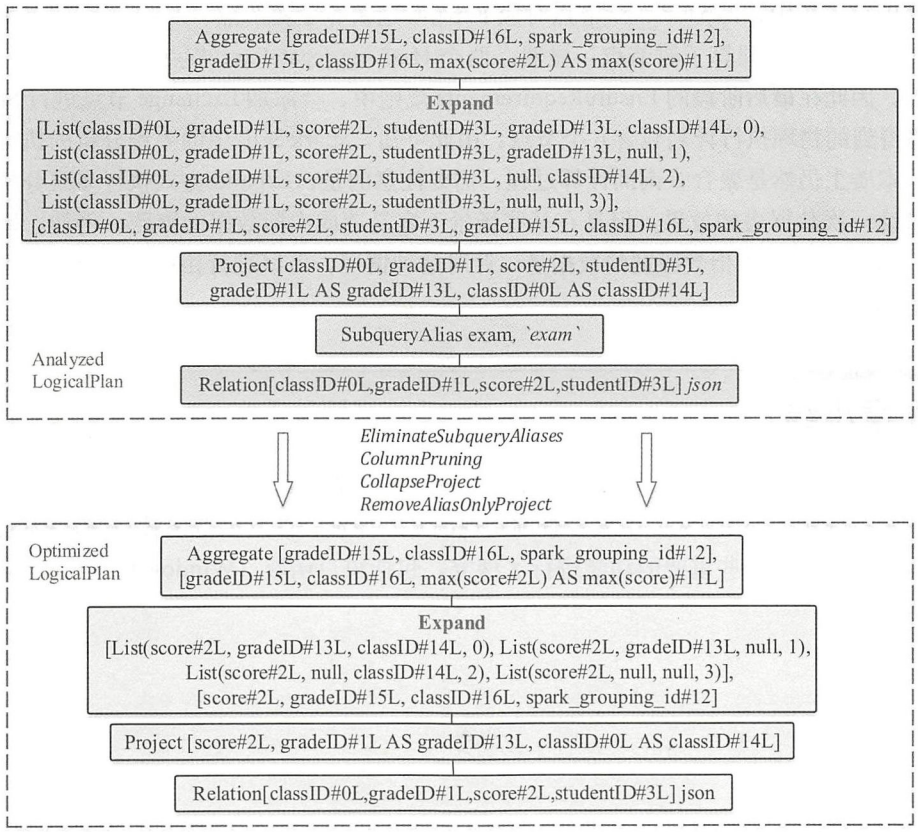


图 7.31 多维分析逻辑算子树（Optimized LogicalPlan）

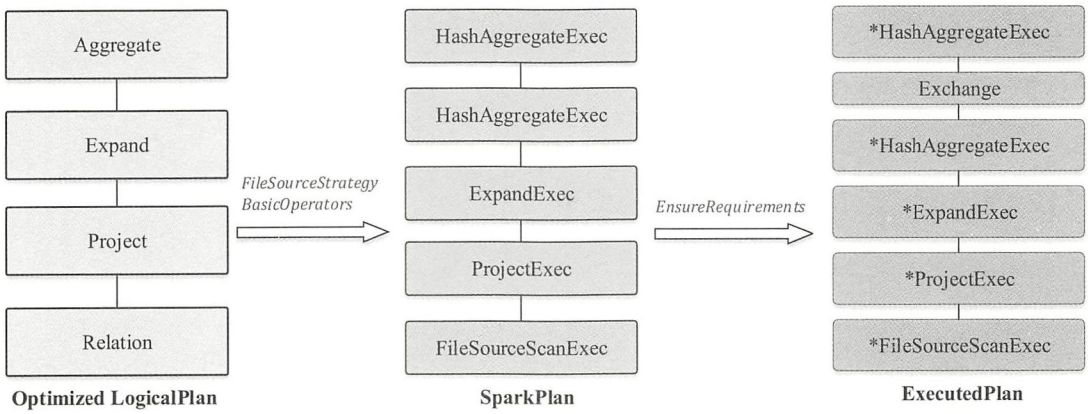


图 7.32 多维分析执行计划生成

在物理算子树中，ExpandExec 节点对应 Expand 逻辑算子树节点，Aggregate 节点生成两个 HashAggregateExec 物理算子树节点。显然，两个 HashAggregateExec 节点之间需要进行分区方面的处理，因此在最后阶段的 EnsureRequirements 规则中，会添加 Exchange 节点进行 Shuffle 操作，最终得到的物理执行计划包含 6 个节点。由此可知，Spark SQL 中的多维分析的执行没有特殊之处，本质上仍然是聚合查询的计算过程。需要注意的是，Expand 方式执行多维分析虽然能够达到只读一次数据表的效果，但是在某些场景下容易造成中间数据的膨胀。例如，数据的维度太高，Expand 会产生指数级别的数据量。针对这种情况，可以进行相应的优化，详细的分析见本书第 11 章。

7.6 本章小结

聚合查询是数据分析领域非常重要的组成部分，也是业界各种主流 SQL 引擎中必不可少的功能。本章分析了 Spark SQL 中聚合查询实现的基本原理，涉及聚合函数与聚合执行的机制。同时，在此基础上对一些重要的功能进行了探究，包括窗口函数（Window Function）和 OLAP 多维分析功能。

第8章

Spark SQL 之 Join 实现

在数据分析应用中，Join 一直以来都是常用的算子，可以将多个关系数据表按照一定的条件连接在一起。对于大数据 SQL 查询引擎来讲，高效地执行 Join 至关重要。在分布式集群环境下，因为涉及数据在不同节点间 Shuffle，Join 可以算是代价最昂贵的操作，背后的实现原理也最复杂。本章对 Spark SQL 中 Join 操作的内部机制进行深入剖析，以帮助读者了解其基本实现方式。

8.1 Join 查询概述

熟悉关系代数的读者应该知道，在数据库设计阶段通常都会遵循一定的范式，以减少数据模型的冗余。这种设计理念对于 OLTP（在线事务处理）类型的应用非常友好，但是在面对多表数据分析的需求时，必须将多个分散的数据表关联起来，也就是需要进行 Join 操作。在 ANSI SQL 标准中，共有 5 种 Join 方式：内连接（Inner）、全外连接（FullOuter）、左外连接（LeftOuter）、右外连接（RightOuter）和交叉连接（Cross）。

Join 查询在时间和空间上都有着很高的复杂度，执行代价比较大，有时还会面临“数据倾斜”等棘手的情况。从 20 世纪 80 年代起，关于 Join 的优化一直都是业界和学术界研究的重点。例如，近年来兴起的“NoSQL 运动”，主要理念之一就是鼓励数据分析人员通过设计“宽表”来直接避免 Join 查询。

本章仍然通过一个简单的例子，从细节层面分析 Join 的完整实现流程。这里将 Join 查询中涉及的关系数据表称为“基本表”，假设除 student 基本表外，还有一个记录了学生成绩的基本表 exam，包含 studentId 和 score 两列。如果想要知道每个学生的姓名（name）和考试成绩（score），在查询层面就需要对 student 表和 exam 表进行连接操作，对应的 SQL 查询语句如下：

```
select name, score from student join exam on student.id = exam.studentId
```

8.2 文法定义与抽象语法树

在 Spark SQL 的 ANTLR 4 文法文件中，与 Join 相关的文法定义如下。可以看到，Join 语句主要针对的是关系数据表，一般处于 From 子语句中。

```
fromClause : FROM relation (',' relation)* lateralView* ;
relation : relationPrimary joinRelation* ;
joinRelation : (joinType) JOIN right=relationPrimary joinCriteria?
         | NATURAL joinType JOIN right=relationPrimary ;
joinType : INNER? | CROSS | LEFT OUTER? | LEFT SEMI | RIGHT OUTER? | FULL OUTER? | LEFT? ANTI ;
joinCriteria : ON booleanExpression | USING '(' identifier (',' identifier)* ')';
```

该文法有几个关键的信息：在 FROM 关键字表示的数据源中，至少包含一个或多个 relation，以及可能的 lateralView（注：本章不涉及该关键字，读者可自行研究）。每个 relation 包含一个主要的数据表（relationPrimary）和零个或多个参与 Join 操作的数据表（joinRelation）。在 joinRelation 中，除参与 Join 的数据表外，比较重要的关键字是 Join 的类型（joinType）和 Join 的条件（joinCriteria）。

目前，Spark SQL 中支持的 Join 类型主要包括 Inner、FullOuter、LeftOuter、RightOuter、LeftSemi、LeftAnti 和 Cross 共 7 种，对应的关键字如表 8.1 所示。

表 8.1　Spark SQL 支持的 Join 类型

查询关键字	Join 类型
inner	Inner
outer | full | fullouter	FullOuter
leftouter | left	LeftOuter
rightouter | right	RightOuter
leftsemi	LeftSemi
leftanti	LeftAnti
cross	Cross

这些不同类型的 Join 操作，可以很方便地用集合的形式进行可视化描述。如图 8.1 所示，分别用 A 和 B 表示参与 Join 操作的两个基本表，Inner 类型的 Join 等价于 A 和 B 参与 Join 的列数据集合求交集，Outer 类型的 Join 等价于 A 和 B 参与 Join 的列数据集合求并集，而 Cross 类型的 Join 对应 A 与 B 之间的笛卡儿积。需要特别注意 Join 操作得到的结果，Left 与 Right 在基本表的全部数据的基础上返回 Join 数据，Semi 跟 Inner 类似，区别在于 Semi 只返回基本表中的列。这里可以通过改变 SQL 的写法来理解，例如：

```
Q1: select student.id from student left semi join exam on student.id = exam.studentId
```

等价于

```
Q2: select id from student where id in (select studentId from exam)
```

Anti Join 的结果与 Semi Join 的结果相反。

```
Q3: select student.id from student left anti join exam on student.id = exam.studentId
```

等价于

```
Q4: select id from student where id not in (select studentId from exam)
```

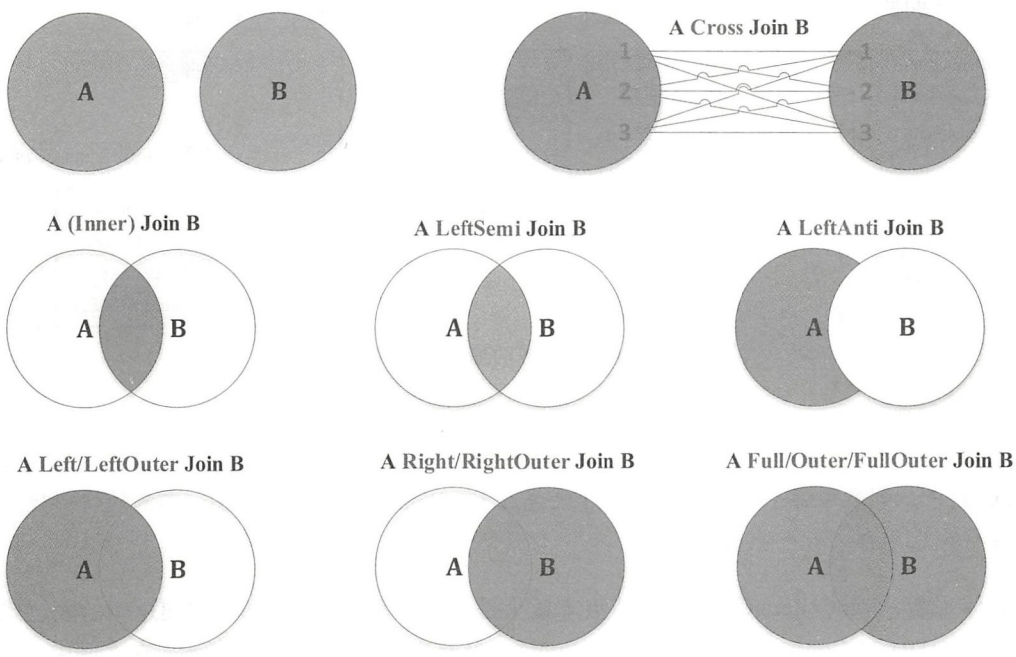


图 8.1 不同的 Join 类型

至于 Join 的条件 joinCriteria 的内容，从文法定义中也可以看到，有 ON 和 USING 关键字两种写法，在实际使用中 ON 较常用。

回到前面例子中的查询语句，经过 ANTLR 4 编译器的处理，该查询语句会生成图 8.2 所示的抽象语法树。类似常见的查询语句，QuerySpecificationContext 节点的子节点 NamedExpres-

sionSeqContext 对应 Select 语句中所选择的列，图 8.2 中的两个 NamedExpressionContext 子节点分别代表 name 和 score 列。对于 Join 查询，值得关注的是 FromClauseContext 节点。对于上述查询，图 8.2 中的第一个 TableNameContext 子节点对应文法定义中的 relationPrimary，即 student 数据表；第二个 TableNameContext 子节点对应 exam 数据表，在 JoinRelationContext 节点下还包含对应 Join 类型的 JoinTypeContext 子节点和对应 Join 条件的 JoinCriteriaContext 子节点。

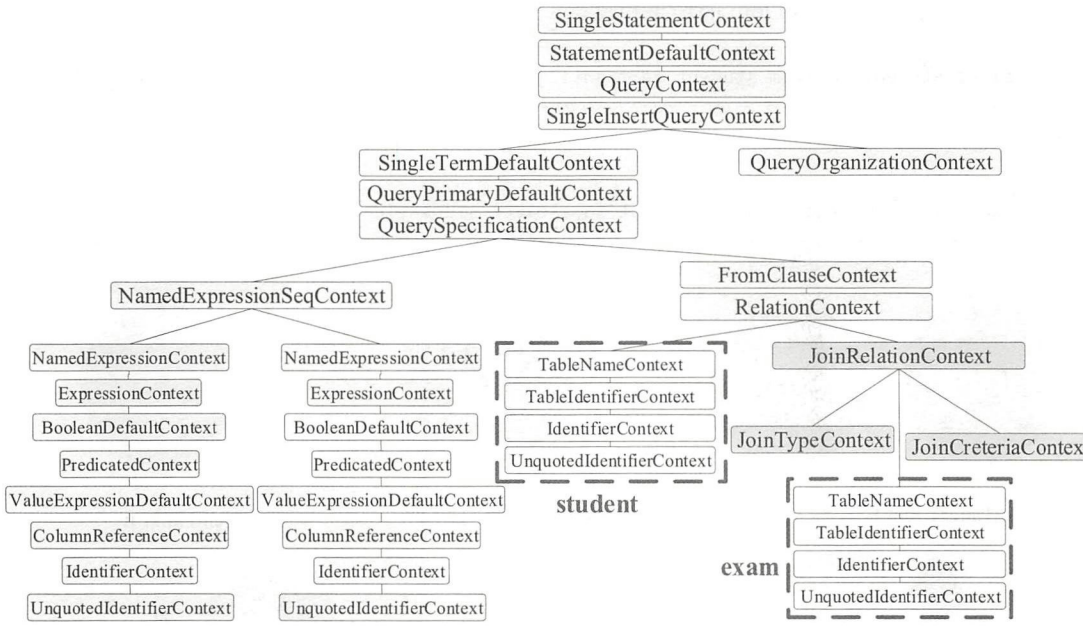


图 8.2　Join 查询生成的抽象语法树（AST）

Note：实际上，在上述语法树对应的数据结构中，FromClauseContext 节点下包含的是 RelationContext 列表，列表中每个 RelationContext 又包含 JoinRelationContext 列表。本例的情形较为特殊，每个列表中都只有一个节点。

具体来看，JoinCriteriaContext 子节点本质上是一个表示 True 和 False 谓词逻辑的表达式节点（BooleanDefaultContext）。JoinCriteriaContext 子节点内容展开如图 8.3 所示。在本例中，该表达式的左、右子表达式分别为“student.id”和“exam.studentId”，这两个表达式都设置了数据表名，属于 DereferenceContext 类型。图 8.3 中的 ComparisonOperatorContext 节点对应列之间的相等关系。

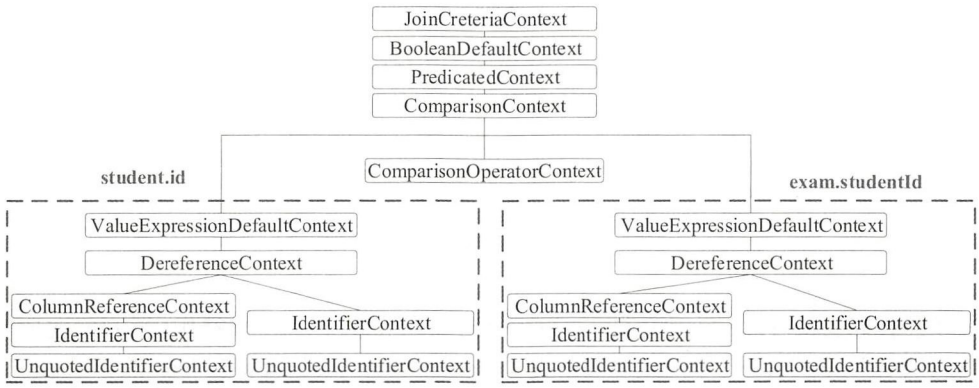


图 8.3 Join 条件细节

8.3 Join 查询逻辑计划

本节分析 Join 查询的逻辑计划阶段。逻辑计划阶段的开始仍然由 AstBuilder 将抽象语法树（AST）生成 Unresolved LogicalPlan，然后在此基础上经过解析得到 Analyzed LogicalPlan，最后经过优化得到 Optimized LogicalPlan。

8.3.1 从 AST 到 Unresolved LogicalPlan

与 Join 算子相关的部分主要在 From 子句中，这里不再展开介绍 Select 语句对应的逻辑算子树生成过程，将视角聚焦在与 Join 相关的逻辑上。具体来看，逻辑计划生成过程由 AstBuilder 类定义的 visitFromClause 方法开始，其核心代码如下。

```
override def visitFromClause(ctx: FromClauseContext): LogicalPlan = withOrigin(ctx) {
    val from = ctx.relation.asScala.foldLeft(null: LogicalPlan) { (left, relation) =>
        val right = plan(relation.relationPrimary)
        val join = right.optionalMap(left)(Join(_, _, Inner, None))
        withJoinRelations(join, relation)
    }
    ctx.lateralView.asScala.foldLeft(from)(withGenerate)
}
```

从 FromClauseContext 中得到的 relation 是 RelationContext 的列表。根据前面的文法分析可知，每个 RelationContext 代表一个通过 Join 连接的数据表集合，每个 RelationContext 中有一个主要的数据表（RelationPrimaryContext）和多个需要 Join 连接的表（JoinRelationContext），如图 8.4 所示。

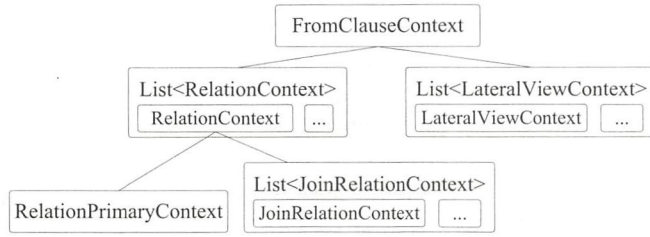


图 8.4　FromClauseContext 的结构

从上述代码逻辑可知，针对 RelationContext 对象列表进行 foldLeft 操作，将已经生成的逻辑计划与新的 RelationContext 中的主要数据表（relationPrimary）结合得到 Join 算子（optionalMap 方法），然后将生成的 Join 算子加入新的逻辑计划中。

对于本例来说，FromClauseContext 对应的 RelationContext 列表中只有一个元素，其 relationPrimary 为 student 数据表。由于初始的 LogicalPlan 为 null，所以上述代码中的 join 值同样为 student 对应的 LogicalPlan。然后调用 withJoinRelations 方法，将得到的 LogicalPlan 与数据表 exam 进行 Join 操作。

在进一步考察 withJoinRelations 的实现逻辑之前，有必要介绍几个重要的对象。首先，用来表示 Join 操作类型的 JoinType，其 UML 如图 8.5 所示。JoinType 实现为抽象类，里面定义了返回 Join 类型字符串的函数。在 Spark 2.1 版本中共实现了 11 个子类，其中 InnerLike 又细分为 Inner 和 Cross 类型。

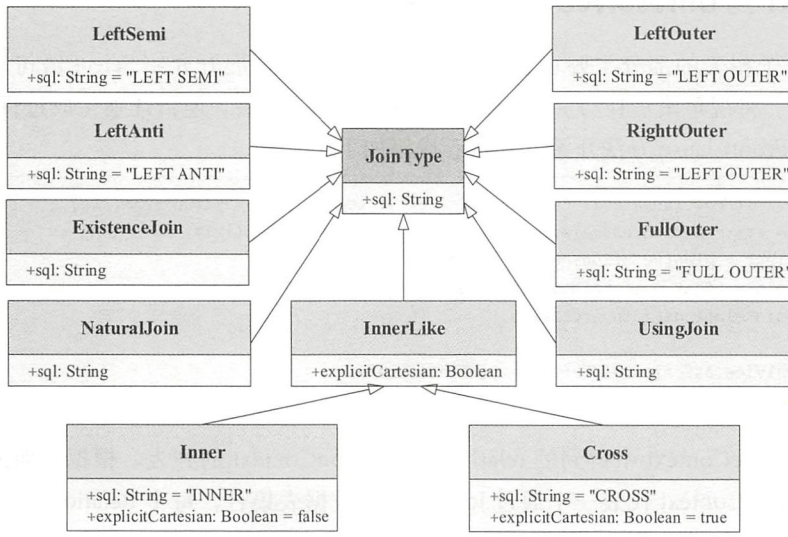


图 8.5　JoinType 的 UML

其次，用来表示 Join 操作的逻辑算子节点是 Join 类，其内部元素如图 8.6 所示，其中 left、right、joinType 和 condition 作为构造参数，分别表示 Join 操作的左节点、右节点、Join 类型和 Join 条件。作为 BinaryNode 节点类型，Join 类中重载了 resolved 属性和 statistics 属性值，同时也重载了 output 函数和 validConstraints 函数，这两个函数逻辑决定了不同 Join 类型输出的列和内部约束条件的整合。这里有必要注意一下 duplicateResolved 函数，在 Join 操作中涉及两个数据表，因此可能存在相同 ID 的表达式，duplicateResolved 函数用来确保不会因为这种重复而出现表达式歧义。

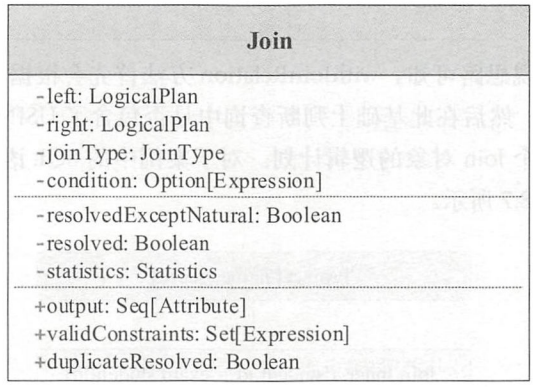


图 8.6 Join 逻辑算子节点

基于上述相关的定义，接下来继续回到 AstBuilder 中的逻辑，在 visitfromClauseContext 方法中生成 primaryRelation（student 表）对应的 LogicalPlan 之后，进入 withJoinRelation 方法中对 JoinRelationContext 中的表进行处理，其实现逻辑如下。

```
private def withJoinRelations(base: LogicalPlan, ctx: RelationContext): LogicalPlan = {
    ctx.joinRelation.asScala.foldLeft(base) { (left, join) =>
      withOrigin(join) {
        val baseJoinType = join.joinType match {
          case null => Inner
          case jt if jt.CROSS != null => Cross
          case jt if jt.FULL != null => FullOuter
          case jt if jt.SEMI != null => LeftSemi
          case jt if jt.ANTI != null => LeftAnti
          case jt if jt.LEFT != null => LeftOuter
          case jt if jt.RIGHT != null => RightOuter
          case _ => Inner
        }
        val (joinType, condition) = Option(join.joinCriteria) match {
          case Some(c) if c.USING != null =>
            (UsingJoin(baseJoinType, c.identifier.asScala.map(_.getText)), None)
          case Some(c) if c.booleanExpression != null =>
```

```
            (baseJoinType, Option(expression(c.booleanExpression)))
          case None if join.NATURAL != null =>
            if (baseJoinType == Cross) {
              throw new ParseException( "NATURAL CROSS JOIN is not supported" , ctx)
            }
            (NaturalJoin(baseJoinType), None)
          case None => (baseJoinType, None)
        }
        Join(left, plan(join.right), joinType, condition)
      }
    }
}
```

根据上述代码的实现思路可知，withJoinRelation 方法首先会根据 SQL 语句中的 Join 类型构造基础的 JoinType 对象，然后在此基础上判断查询中是否包含了 USING 等关键字，并进行进一步的封装，最终得到一个 Join 对象的逻辑计划。对于案例中的 SQL 语句，最终生成的逻辑计划实际上非常简单，如图 8.7 所示。

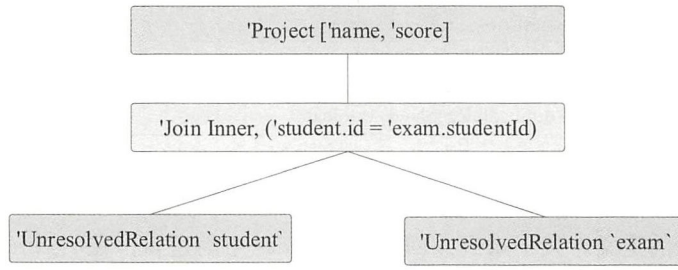


图 8.7　Unresolved LogicalPlan 生成

8.3.2 从 Unresolve LogicalPlan 到 Analyzed LogicalPlan

在 Analyzer 中，与 Join 相关的解析规则有很多，包括 ResolveReferences 和 ResolveNaturalAndUsingJoin 等。在介绍这些规则的同时，下面的分析仍然以案例为主。对于图 8.7 中的逻辑算子树，整个解析过程比较简单，如图 8.8 所示。

可以看到，ResolveRelations 和 ResolveReferences 两条规则产生了影响。ResolveRelations 规则的作用是从 Catalog 中找到 student 和 exam 的基本信息，包括数据表存储格式、每一列列名和数据类型等。ResolveReferences 规则负责解析所有列信息，对于上面的逻辑算子树，ResolveReferences 的解析是一个自底向上的过程，将所有 UnresolvedAttribute 与 UnresolvedExtractValue 类型的表达式转换成对应的列信息。

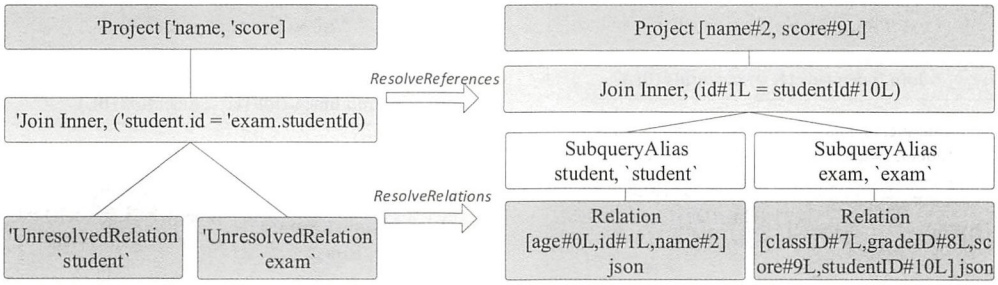


图 8.8 Resolved LogicalPlan 生成

在 ResolveReferences 规则中，如果传入的逻辑算子树根节点为 Join 类型，则还存在如下的逻辑来处理 Join 中冲突的列。在 dedupRight 方法中，针对存在冲突的表达式会创建一个新的逻辑计划，通过增加别名（Alias）的方式来避免列属性的冲突。

```
private def dedupRight (left: LogicalPlan, right: LogicalPlan): LogicalPlan = {
    val conflictingAttributes = left.outputSet.intersect(right.outputSet)
    ......
}
case j @ Join(left, right, _, _) if !j.duplicateResolved =>
    j.copy(right = dedupRight(left, right))
```

根据该逻辑，如果 Join 操作存在重名的属性（左、右子节点的输出属性名集合有重叠），那么就调用 dedupRight 方法将右子节点对应的 Expression 用一个新的 Expression ID 表示，这样即使出现同名，经过处理之后 Expression ID 也不相同，因此可以区分 Join 操作中不同的数据表。

在 Analyzer 中，还有一个和 Join 操作直接相关的 ResolveNaturalAndUsingJoin 规则。该规则将 NATUAL 或 USING 类型的 Join 转换为普通的 Join。其主要处理逻辑是根据 Join 两边的输出列信息计算得到总的输出列信息，然后将 Project 算子添加到常规的 Join 算子上。

8.3.3 从 Analyzed LogicalPlan 到 Optimized LogicalPlan

经过 Analyzer 的解析得到上述逻辑算子树之后，开始进入逻辑算子树的优化阶段。前面的章节分析过一些常见的优化规则，这里再次熟悉一下与本案例相关的规则。

逻辑算子树优化的第一阶段就是消除多余的别名，对应 EliminateSubqueryAliases 优化规则。其逻辑非常简单，将 SubqueryAlias(_, child, _) 节点直接替换为 child 节点，起到的作用如图 8.9 所示。可以看到，Relation 原来的 SubqueryAlias 父节点已经被移除，Join 成为 Relation 的父节点。

经过别名消除之后，接下来的优化是常用的列剪裁（Column Pruning），在上述逻辑算子树中，父节点只需要用到两个数据表中的 4 列，因此可以在 Relation 节点之后添加新的 Project 节点进行列剪裁的操作。如图 8.10 所示，经过列剪裁优化后，得到新的逻辑算子树。

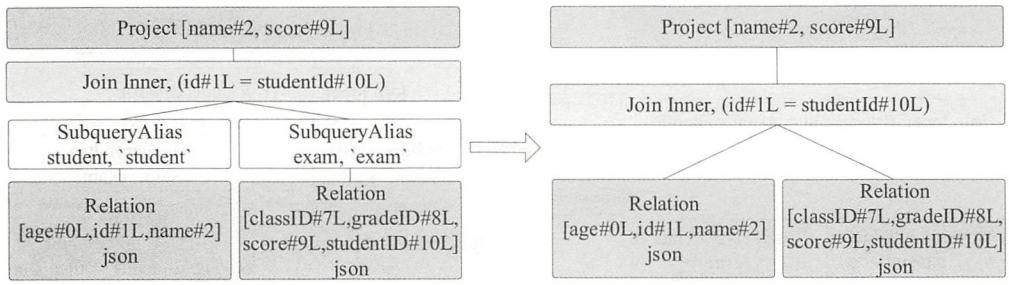


图 8.9　EliminateSubqueryAliases 优化规则

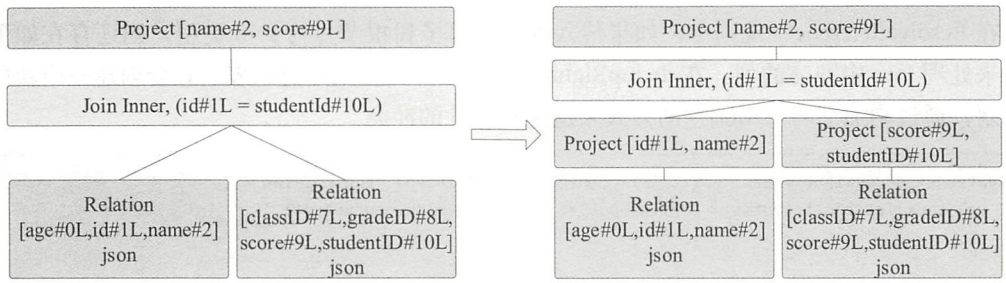


图 8.10　ColumnPruning 优化规则

然后，优化过程将考虑相关算子中的过滤条件。对于 Join 来讲，其连接条件需要保证两边的列都不为 null，因此会触发 InferFiltersFromConstraints 优化规则。如图 8.11 所示，经过该规则的处理，Join 算子中的连接条件多了两个，分别约束 student 表中的 ID 和 exam 表中的 studentID 不为 null。

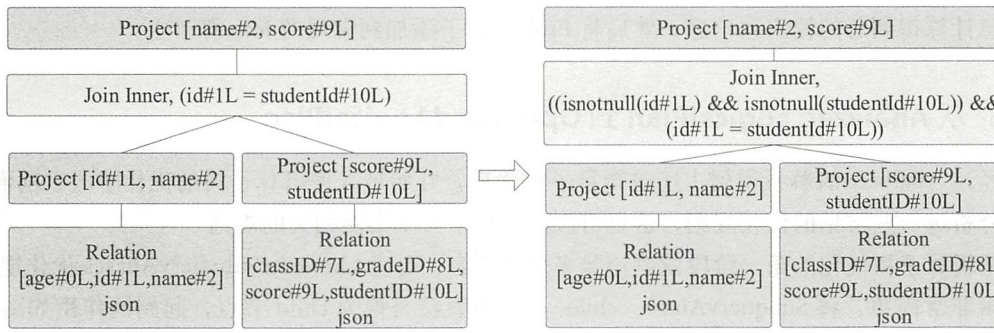


图 8.11　InferFiltersFromConstraints 优化规则

在 Optimizer 阶段，有一条专门针对 Join 算子的 PushPredicateThroughJoin 优化规则。顾名思义，该优化规则所起的作用就是对 Join 中连接条件可以下推到子节点的谓词进行下推操作。

因为经过上一步的优化规则，逻辑算子树中 Join 节点多了两个条件用来判定列不为 null，这两个条件只涉及单个数据表，因此可以下推到对应的子节点中，这样可以达到尽早过滤数据的效果。经过 PushPredicateThroughJoin 优化规则后，得到的逻辑算子树如图 8.12 所示。

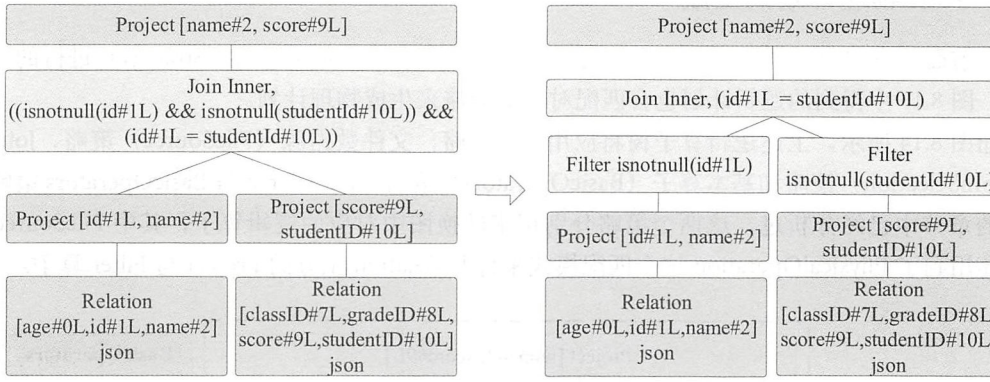


图 8.12 PushPredicateThroughJoin 优化规则

经过 PushPredicateThroughJoin 优化规则后，Join 中的两个连接条件生成了对应的两个 Filter 节点。一般来讲，优化阶段会将过滤条件尽可能地下推，因此逻辑算子树中的 Filter 节点还会被继续处理。该逻辑对应 PushDownPredicate 优化规则，得到的新的逻辑算子树如图 8.13 所示，Filter 节点已经位于 Project 节点之下。

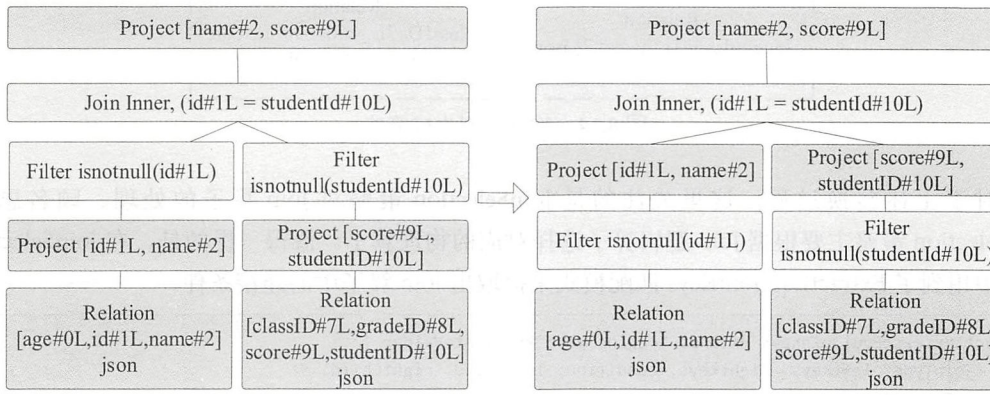


图 8.13 PushDownPredicate 优化规则

至此，整个逻辑算子树的优化工作完成。整个过程应用了 5 条优化规则，对应了 5 次树型结构的操作转换。从上述转换过程来看，对于 Join 节点，密切相关的是 InferFiltersFromConstraints 和 PushPredicateThroughJoin 两条优化规则。

8.4 Join 查询物理计划

8.4.1 Join 物理计划的生成

在前面的章节中提到过，从逻辑计划到物理计划的生成是基于策略（Strategy）进行的。同样的，图 8.13 中得到的逻辑计划也会匹配对应的策略来生成物理计划。

如图 8.14 所示，上述逻辑算子树将应用 3 个策略：文件数据源（FileSource）策略、Join 选择（JoinSelection）策略和基本算子（BasicOperators）策略。FileSource 与 BasicOperators 策略在前面的章节中已经分析过，这两个策略分别用来转换图中对应的逻辑算子，其中 FileSource 策略中还用到了 PhysicalOperation 这个匹配模式来合并 Relation 上方的 Project 与 Filter 算子。

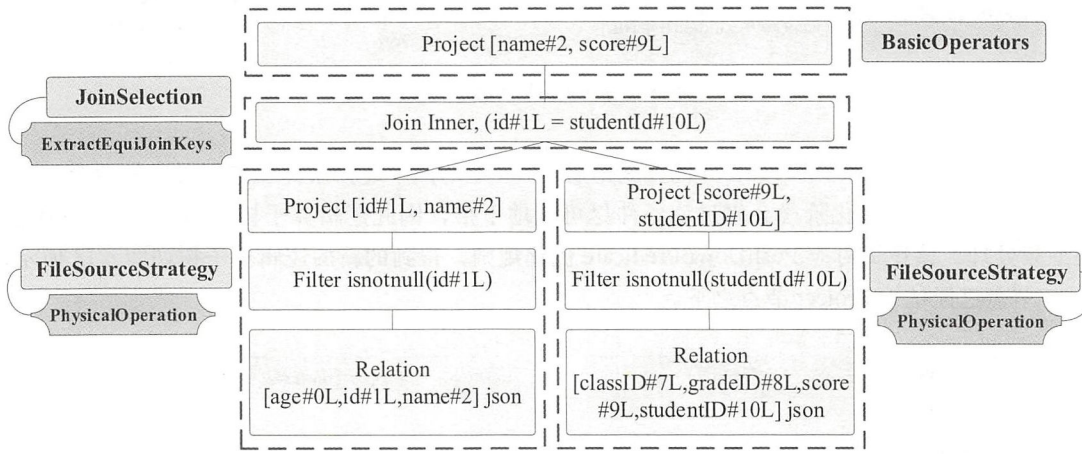


图 8.14　物理计划生成策略

对于上述转换过程，这里关注的是 JoinSelection 策略对 Join 算子的处理。顾名思义，JoinSelection 策略主要根据 Join 逻辑算子选择对应的物理算子。值得一提的是，在 JoinSelection 策略中用到了 ExtractEquiJoinKeys 匹配模式来提取出 Join 算子中的连接条件。

```
object ExtractEquiJoinKeys extends Logging with PredicateHelper {
/** (joinType, leftKeys, rightKeys, condition, leftChild, rightChild) */
type ReturnType =
  (JoinType, Seq[Expression], Seq[Expression], Option[Expression], LogicalPlan, LogicalPlan)
def unapply(plan: LogicalPlan): Option[ReturnType] = ....
}
```

具体来讲，ExtractEquiJoinKeys 模式的主要逻辑如下：如果是等值连接（Equi-Join），则将左、右子节点的连接 key 都提取出来。此时存在两种情况：EqualTo 和 EqualNullSafe。两者的区

别在于对空值（null）是否敏感，其中，EqualTo 对空值是敏感的（对空值没有额外的处理逻辑），EqualNullSafe 在一般情况下的处理逻辑基本与 EqualTo 一样，但是它会对空值做处理，即赋予相应类型的默认值。这两种情况实际上和用户编写的 SQL 语句有关，当 SQL 语句中 Join 条件表达式为“=”或“==”时，会对应 EqualTo 模式；当 Join 条件表达式为“<=>”时，会对应 EqualNullSafe 模式。此外，在 ExtractEquiJoinKeys 中还通过 otherPredicates 记录除 EqualTo 和 EqualNullSafe 类型外的其他条件表达式，这些谓词基本上可以在 Join 算子执行 Shuffle 操作之后在各个数据集上分别处理。

8.4.2 Join 物理计划的选取

基于上面的分析，该案例对应生成的物理计划（SparkPlan）如图 8.15 所示。可以看到，物理计划的节点和逻辑计划的节点基本上一一对应。在生成物理计划的过程中，JoinSelection 根据若干条件判断采用何种类型的 Join 执行方式。目前在 Spark SQL 中，Join 的执行方式主要有 BroadcastHashJoinExec、ShuffledHashJoinExec、SortMergeJoinExec、BroadcastNestedLoopJoinExec 和 CartesianProductExec 这 5 种。

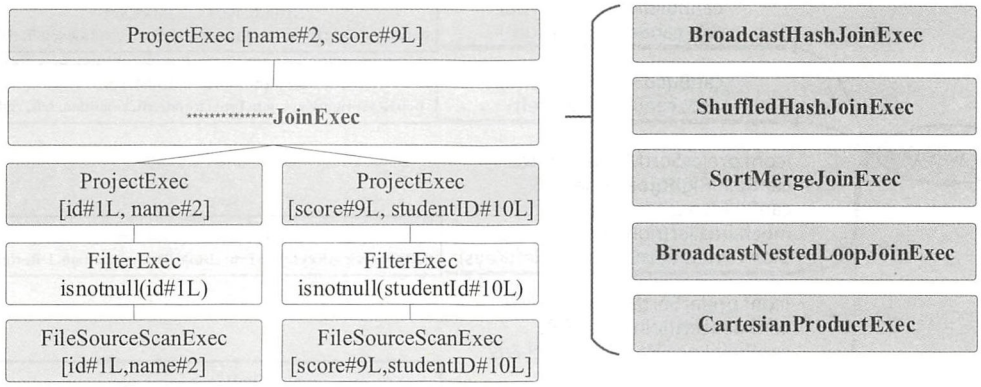


图 8.15 物理计划 SparkPlan 的生成

在分析 JoinSelection 的具体逻辑之前，先介绍 Join 实现中会用到的一些基本概念和针对数据表的一些特点的定义。

- 数据表能否广播：在两个表的 Join 操作中，如果一个数据表的数据量非常小，则可以将这个表广播到另一个表数据所在的所有节点上。在 JoinSelection 中通过 canBroadcast 方法来判断一个数据表对应的逻辑计划能否广播。在 Spark SQL 中，可以通过“spark.sql.autoBroadcastJoinThreshold”参数设置自动广播的阈值（单位为 Byte），当某个表的数据量小于这个阈值时，这个表将自动进行广播操作，该参数默认值为 10MB。

- Join 操作的 BuildSide：参与 Join 操作的左右两个数据表起到的作用是不一样的，例如，在 BroadcastHashJoin 中需要决定广播哪个数据表等。这里的 BuildSide 可以简单理解为“构建的一边”，其具体含义会在后面的执行阶段分析。目前，在 Catalyst 中 BuildSide 作为一个抽象类，包含 BuildLeft 和 BuildRight 两个子类，一般在构造 Join 的执行算子时，都会传入一个 BuildSide 的构造参数。在 JoinSelection 中通过 canBuildRight 和 canBuildLeft 判断一个 Join 类型能否“构建”右表和左表，根据源码可知，只有 InnerLike 或 RightOuter 类型的 Join 时，左表才能够被“构建”。
- 建立 HashMap（BuildLocalHashMap）：某些 Join 在执行过程中需要创建 HashMap 以在内存中保存相关的数据，当数据量大的时候，单个分区上创建 HashMap 可能导致内存溢出。在 JoinSelection 中实现了一个比较粗粒度的方法 canBuildLocalHashMap 来判断某个逻辑计划能否满足创建本地 HashMap 的条件，主要思想是当前逻辑计划的数据量小于数据广播阈值与 Shuffle 分区数目的乘积。

接下来，就是 JoinSelection 中的具体判断逻辑了。如图 8.16 所示，对于 5 种不同类型的 Join 执行方式，JoinSelection 中有着先后的匹配逻辑。

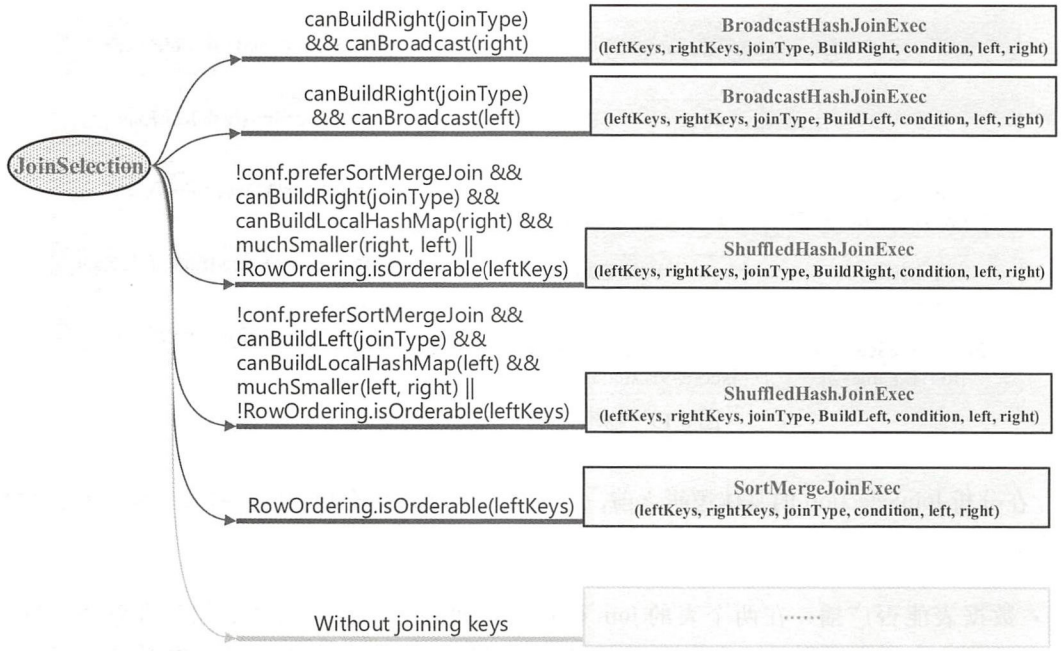


图 8.16　Join 物理计划选取逻辑 JoinSelection

优先级最高的是 BroadcastHashJoinExec，这种 Join 执行方式相对来讲效率最高，因此也是最先进行判断的。具体来讲，它包含两种情况。

- 能够广播右表（canBroadcast）且右表能够“构建”（canBuildRight），那么构造参数中传入的是 BuildRight。
- 能够广播左表（canBroadcast）且左表能够“构建”（canBuildLeft），那么构造参数中传入的是 BuildLeft。

优先级次之的是 ShuffledHashJoinExec，从图 8.16 中可以看到，这种类型的 Join 执行方式需要满足多种条件。ShuffledHashJoinExec 的构造同样分为 BuildLeft 和 BuildRight 两种情况，这里以 BuildRight 为例。首先，配置中优先开启 SortMergeJoin 的参数（“spark.sql.join.preferSortMergeJoin”）并设置为 false，且右表需要满足能够“构建”（canBuildRight）和能够建立 HashMap（canBuildLocalHashMap），同时右表的数据量要比左表的数据量小很多（3 倍以上）。此外，还有一种生成 ShuffledHashJoinExec 的情况是参与连接的 key 不具有排序的特性。

最常见的 Join 执行方式就是 SortMergeJoinExec，从图 8.16 中可以看到，参与 Join 的 key 满足可排序的特性即可。所以，在实际生产环境中，绝大部分 Join 的执行都是采用 SortMergeJoinExec 方式进行的。

剩下的情况都是不包含 Join 条件的语句了，大致逻辑如下：首先判断是否执行数据表广播操作，对应 BuildLeft 和 BuildRight 两种情况，生成 BroadcastNestedLoopJoinExec 类型的 Join 物理算子。如果不满足数据表广播操作，而 Join 类型是 InnerLike，那么就会生成 CartesianProductExec 类型的 Join 物理算子。如果上述情况都不满足，那么只能选择两个数据表中数据量相对较少的数据表来做广播，同样生成 BroadcastNestedLoopJoinExec 类型的 Join 物理算子。

8.5 Join 查询执行

经过前面的步骤，最终将生成特定类型的 Join 物理算子。本节介绍 Join 查询的最后执行阶段，仍然基于案例分析各种执行方式涉及的技术细节。

8.5.1 Join 执行基本框架

在分析不同类型 Join 的具体执行过程之前，先介绍 Join 执行的基本框架。框架中的一些概念和定义在不同的 Join 场景下是通用的。

在 Spark SQL 中，Join 的实现都基于一个基本的流程，如图 8.17 所示。根据角色的不同，参与 Join 操作的两张表分别被称为“流式表（StreamTable）”和“构建表（BuildTable）”，不同表的角色在 Spark SQL 中会通过一定的策略进行设定。通常来讲，系统会默认将大表设定为流式表，将小表设定为构建表。流式表的迭代器为 streamedIter，构建表的迭代器为 buildIter。遍历 streamedIter 中每条记录，然后在 buildIter 中查找相匹配的记录。这个查找过程称为 Build 过程，

每次 Build 操作的结果为一条 JoinedRow（A, B），其中 A 来自 streamedIter，B 来自 buildIter，这个过程为 BuildRight 操作；而如果 B 来自 streamedIter，A 来自 buildIter，则为 BuildLeft 操作。

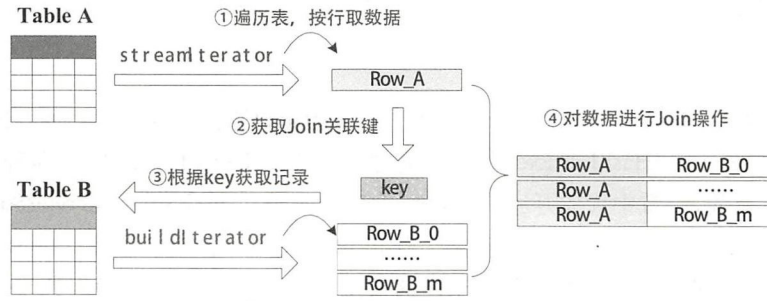


图 8.17　Join 操作的基本流程

对于 LeftOuter、RightOuter、LeftSemi 和 LeftAnti，它们的 Build 类型是确定的，即 LeftOuter、LeftSemi、LeftAnti 为 BuildRight，RightOuter 为 BuildLeft 类型。对于 Inner，BuildLeft 和 BuildRight 两种都可以，选择不同，可能有着很大的性能区别。

在具体的 Join 实现层面，Spark SQL 提供了 BroadcastJoinExec、ShuffleHashJoinExec 和 SortMergeJoinExec 这 3 种机制，以下 3 个小节分别进行介绍。

8.5.2 BroadcastJoinExec 执行机制

该 Join 实现的主要思想是对小表进行广播操作，避免大量 Shuffle 的产生。这也是一种常见的思路，例如，在数据仓库的常见模型中（比如星型模型或雪花模型），数据表一般分为两种：事实表和维度表。维度表一般指固定的、变动较少的表，例如联系人、物品种类等，一般数据有限。而事实表一般记录流水，比如销售清单等，通常随着时间的延长不断膨胀。

Join 操作是对两个表中 key 值相同的记录进行连接，在 Spark SQL 中，对两个表做 Join 操作最直接的方式是先根据 key 分区，然后在每个分区中把 key 值相同的记录提取出来进行连接操作。这种方式不可避免地涉及数据的 Shuffle，而 Shuffle 在各个大数据计算框架中都是比较耗时的操作。因此，当一个大表和一个小表进行 Join 操作时，为了避免数据的 Shuffle，可以将小表的全部数据分发到每个节点上，供大表直接使用。

在 Spark SQL 中，BroadcastJoinExec 可以进行用户设置，触发 BroadcastJoinExec 的场景有两个。

- 被广播的表需要小于参数（spark.sql.autoBroadcastJoinThreshold）所配置的值，默认是 10MB。
- 在 SQL 语句中人为地添加了 Hint（MAPJOIN、BROADCASTJOIN 或 BROADCAST）。

需要注意的是，在 Outer 类型的 Join 中，基表不能被广播，例如当 A left outer join B 时，只能广播右表 B。一般 BroadcastJoinExec 只适用于广播较小的表，否则数据的冗余传输远大于 Shuffle 的开销。另外，广播时需要将被广播的表读取到 Driver 端，当频繁有广播出现时，对 Driver 端的内存也会造成较大压力。

基于广播的 Join 的物理执行计划和最终执行计划如图 8.18 所示。具体执行过程如图 8.19 所示。BroadcastExchange 将广播表广播到每个节点上进行 Join 操作。

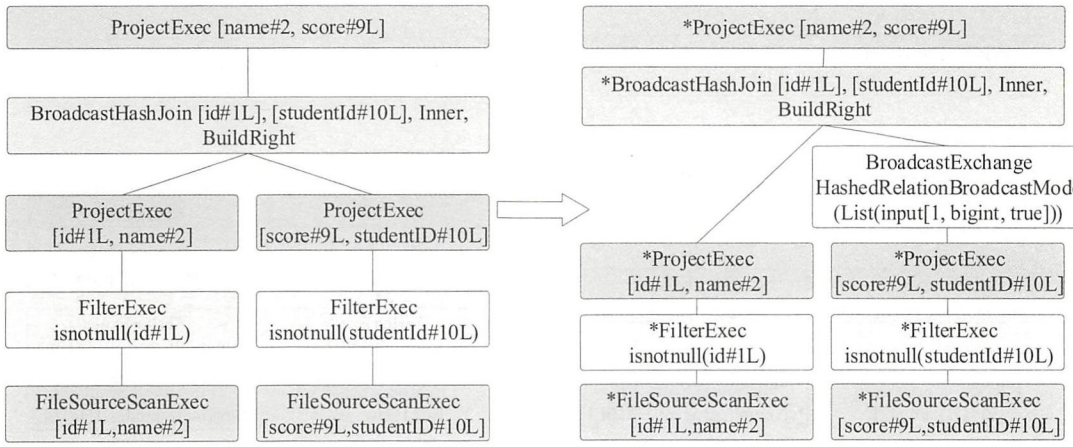


图 8.18 BroadcastJoinExec 物理执行计划和最终执行计划

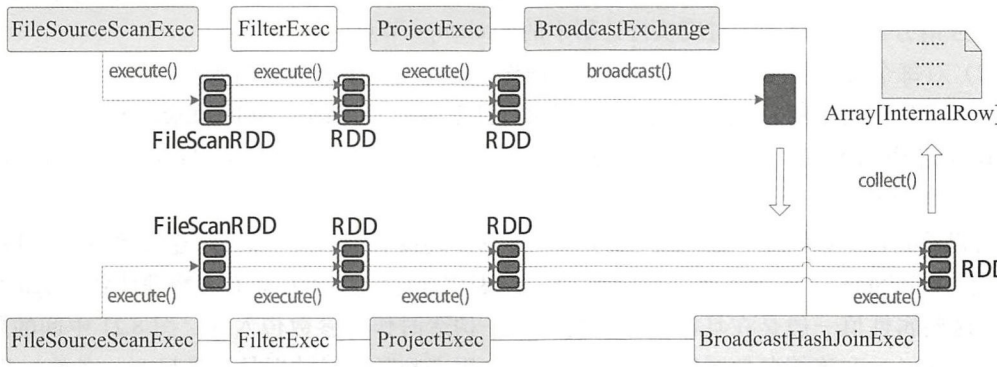


图 8.19 执行过程

8.5.3 ShuffledHashJoinExec 执行机制

前面分析物理计划生成的时候已经知道，生成 ShuffledHashJoinExec 的条件非常苛刻。首先，配置参数 conf.preferSortMergeJoin（spark.sql.join.preferSortMergeJoin），将其设置为 false，确

保优先考虑 ShuffledHashJoinExec 执行方式；其次，需要满足 canBuildLocalHashMap 条件，具体实现为当前表的数据量小于广播数据量阈值与 Shuffle 时配置的 partition 数目（默认为 200）的乘积；最后，满足条件 muchSmaller，一个表的数据量的 3 倍小于或等于另一个表的数据量。

当案例查询满足上面的条件时，其物理执行计划和最终执行计划如图 8.20 所示。根据执行计划可知，ShuffleHashJoinExec 执行机制分为两步。

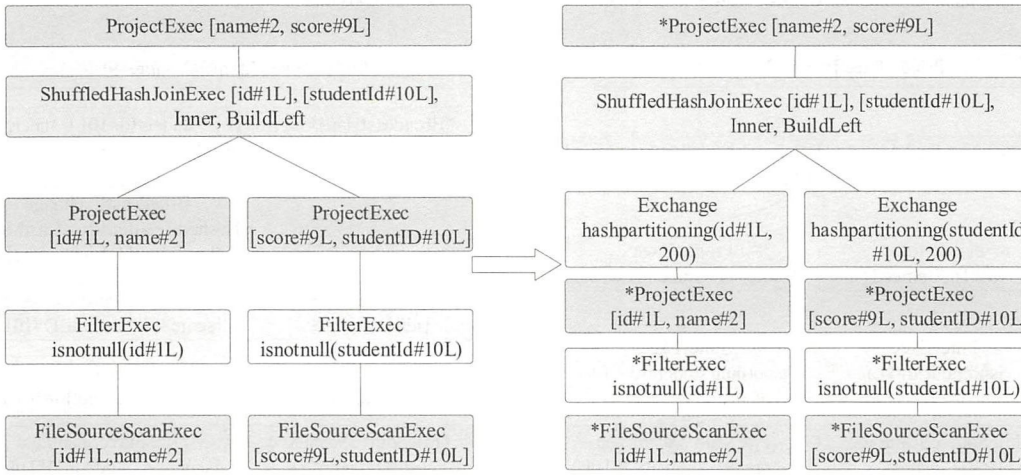


图 **8.20** ShuffledHashJoinExec 物理执行计划和最终执行计划

(1) 对两张表分别按照 Join key 进行重分区，即 Shuffle，目的是为了让有相同 key 值的记录分到对应的分区中，这一步对应执行计划中的 Exchange 节点。

(2) 对每个对应分区中的数据进行 Join 操作，此处先将小表分区构造为一张 Hash 表，然后根据大表分区中记录的 key 值进行匹配，即执行计划中的 ShuffledHashJoinExec 节点。

在介绍 ShuffledHashJoinExec 的实现机制之前，有必要先了解其父类 HashJoin。HashJoin 的操作框架如图 8.21 所示。

可以看到，其基本属性包括连接左键 (leftKeys)、连接右键 (rightKeys)、连接类型 (joinType)、构建侧 (buildSide)、连接条件 (condition)、左表物理计划 (left) 和右表物理计划 (right) 共 7 个，这些属性值一般是在具体的 Join 物理算子构造时作为参数传入的。图 8.21 中间的属性 boundCondition 和连接条件属性 condition 等价，用来判断一行数据是否满足 Join 条件。此外，图 8.21 中还有两个属性对 (buildPlan, streamedPlan) 与 (buildKeys, streamedKeys)，用来区分参与 Join 的两个数据表（构建表和流式表）的角色。构建表在 Join 过程中会创建一个 HashMap，用来支持数据的查找，属于“静态”的一方。流式表在 Join 过程中，一行一行地在构建表对应的 HashMap 中查找数据，属于“动态”的一方。

在 HashJoin 中，已经在构造的时候确定了 BuildSide 是 BuildLeft 还是 BuildRight，因此两个

表的角色也就很容易确定了。通常情况下，会将数据量较小的表作为构建表，将数据量较大的表作为流式表。考虑到 Stream 表属于动态的一方，会涉及数据的分区，因此其 outputPartitioning 由 StreamPlan 中的输出分区决定。

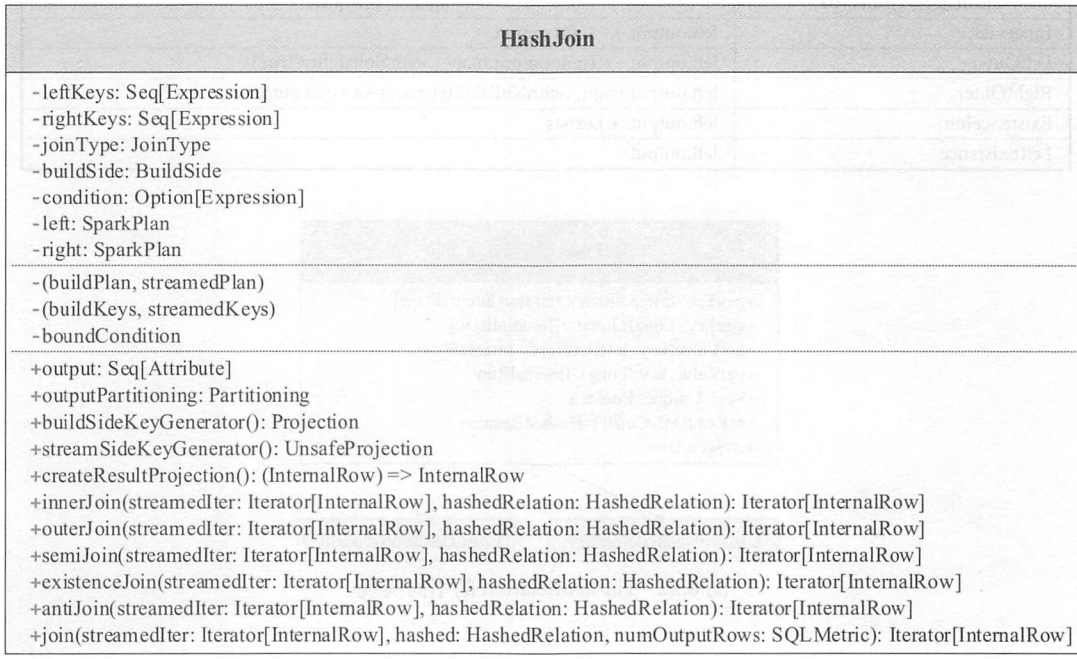


图 **8.21** HashJoin 的操作框架

HashJoin 中的 output 函数表示输出的列，由 Join 类型决定，具体如表 8.2 所示。同样的，在 createResultProjection 方法中，逻辑也和 Join 类型相关。当 Join 类型为 LeftExistence 时，创建的 Projection 的 schema 和 output 相同；否则，采用的 schema 为 streamedPlan 的输出加上 buildPlan 的输出。

在 HashJoin 中最核心的部分就是图 8.21 中的 6 个 Join 函数了，不同类型的 Join 实现逻辑体现在其中。从这些函数的输入参数中可以看到，都包含了一个 HashedRelation 类型。实际上，HashedRelation 就对应了上面提到的构建表，起到了 HashMap 的作用。

如图 8.22 所示，HashedRelation 实际上只是 KnownSizeEstimation 之上的一个接口，该接口支持根据 key 获取匹配到的 InternalRow 迭代器或单行数据。Key 的类型除常见的 InternalRow 外，还支持 Long 数据类型。HashedRelation 具体实现包括 UnsafeHashedRelation 和 LongHashedRelation 两种。UnsafeHashedRelation 是常用的类型，内部依赖 BytesToBytesMap 数据结构，而 LongHashedRelation 类型的 HashedRelation 是一种基于追加方式的 HashMap，其键值对形式为

(Long, UnsafeRow)。HashedRelation 内部实现原理可以暂时跳过，读者在熟悉了第 9 章 Tungsten 技术细节后可自行深入学习。

表 8.2　不同 Join 类型的输出列

Join 类型 (JoinType)	输出列 (output)
InnerLike	left.output ++ right.output
LeftOuter	left.output ++ right.output.map(_.withNullability(true))
RightOuter	left.output.map(_.withNullability(true)) ++ right.output
ExistenceJoin	left.output :+ j.exists
LeftExistence	left.output

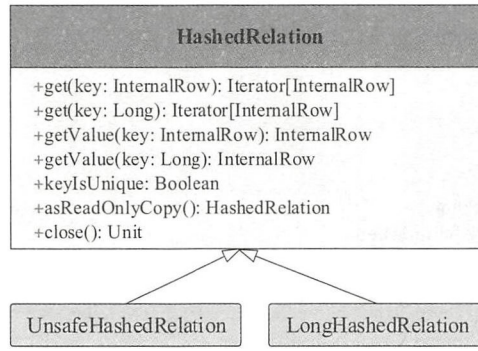


图 8.22　HashedRelation 的不同实现

了解了 HashJoin，理解 ShuffledHashJoinExec 就比较简单了。ShuffledHashJoinExec 是 HashJoin 的子类，其执行过程的实现很简单，核心代码如下。首先，对构建表建立 HashedRelation，然后调用 HashJoin 中的 Join 方法，对当前流式表中的数据行在 HashedRelation 中查找数据进行 Join 操作。

```
protected override def doExecute(): RDD[InternalRow] = {
    streamedPlan.execute().zipPartitions(buildPlan.execute()) { (streamIter, buildIter) =>
      val hashed = buildHashedRelation(buildIter)
      join(streamIter, hashed, numOutputRows)
    }
}
```

8.5.4 SortMergeJoinExec 执行机制

在 Spark SQL 中，SortMergeJoinExec 是 Join 查询的主要实现方式。根据前面几个小节的分析可知，Hash 系列的 Join 实现中都将一侧的数据完全加载到内存中，这对于一定大小的表比较适用。然而，当两个表的数据量都非常大时，无论使用哪种方法都会对计算内存造成很大压力。

通常情况下，特别是当两个表的数据量都非常大时，Spark SQL 会采用 SortMergeJoinExec 的方式来执行。

当案例使用 SortMergeJoinExec 时，其物理执行计划和最终执行计划如图 8.23 所示。

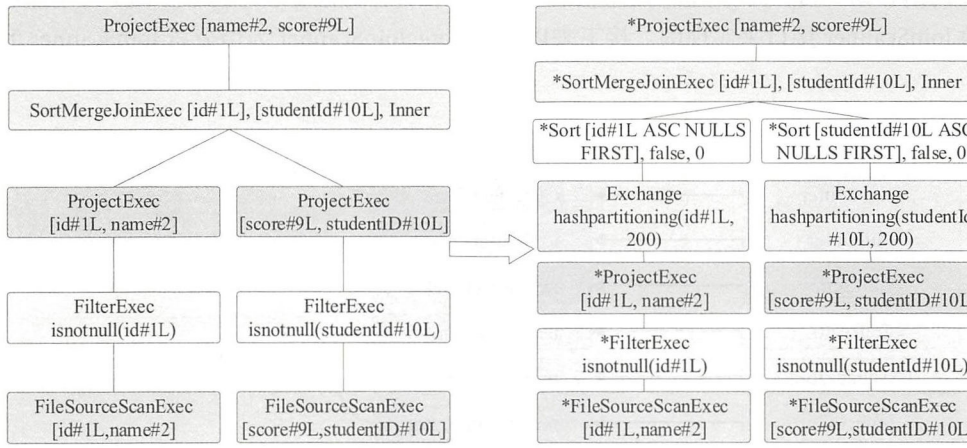


图 8.23 SortMergeJoinExec 物理执行计划和最终执行计划

根据其原理，SortMergeJoinExec 实现方式并不用将一侧数据全部加载后再进行 Join 操作，其前提条件是需要在 Join 操作前将数据排序。如图 8.24 所示，为了让两条记录能连接到一起，需要将具有相同 key 的记录分发到同一个分区，因此一般会进行一次 Shuffle 操作（物理执行计划中的 Exchange 节点），根据 Join 条件确定每条记录的 key，并且基于该 key 进行分区，将可能连接到一起的记录分发到同一个分区中，这样在后续的 Shuffle 读阶段就可以将两个表中具有相同 key 的记录分到同一个分区处理。

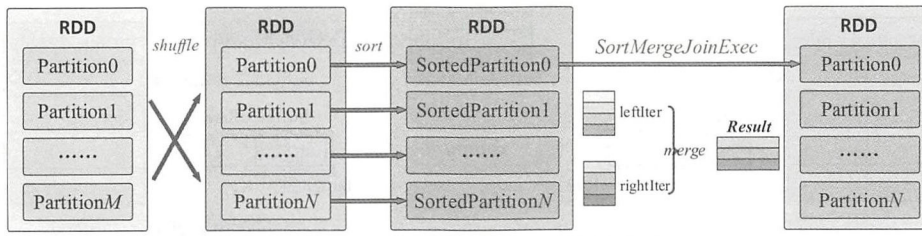


图 8.24 SortMergeJoinExec 执行过程

经过 Exchange 节点操作之后，分别对两个表中每个分区里的数据按照 key 进行排序（图 8.23 中 SortExec 节点），然后在此基础上进行 merge sort 操作。在遍历流式表时，对于每条记录，都采用顺序查找的方式从构建查找表中查找对应的记录。由于排序的特性，每次处理完一条记录后只需要从上一次结束的位置开始继续查找，SortMergeJoinExec 执行时就能够避免大量无用的

操作，对于性能的提升很有帮助。

SortMergeJoinExec 的整体实现如图 8.25 所示，不同的 Join 类型返回不同的 RowIterator 作为 Join 结果。RowIterator 主要实现了 advanceNext 和 getRow 两个方法，其中 advanceNext 是将 Iterator 向前移动一行，而 getRow 用来获取当前行。在具体方法实现中，RowIterator 是通过调用对应的 JoinScanner 接口来实现的。接下来以 SortMergeJoinScanner 为例分析 JoinScanner 的实现细节。

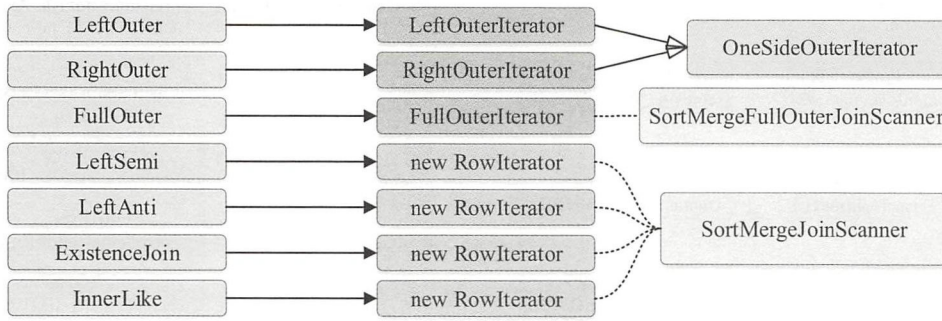


图 8.25　SortMergeJoinExec 的整体实现

SortMergeJoinScanner 是查找匹配数据的核心类，如图 8.26 所示。在 SortMergeJoinScanner 的构造参数中会传递 streamedTable 的迭代器 (streamedIter) 和 bufferedTable 的迭代器 (bufferedIter)，考虑到 streamedTable 与 bufferedTable 都是已经排好序的，因此在匹配满足条件数据的过程中只需要不断移动迭代器，得到新的数据行进行比较即可。在 SortMergeJoinScanner 中，两个表迭代器所指向的数据行分别用 streamedRow 和 bufferedRow 表示。数据行对应的 Join 操作的 key 分别为 streamedRowKey 与 bufferedRowKey，这些对象都属于 InternalRow 类型。

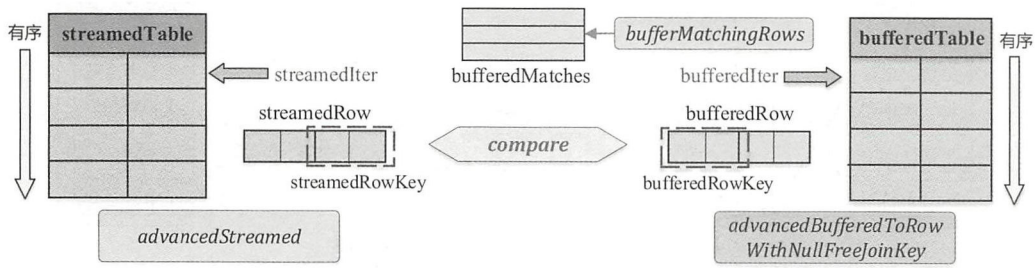


图 8.26　SortMergeJoinScanner 实现

对于 streamedTable 来说，迭代器移动得到新的 streamedRow 由 advancedStreamed 函数完成。该函数返回的 Boolean 值表示 streamedTable 是否还有数据。函数每次调用 streamedIter 执行 advanceNext 操作，重新对 streamedRow 赋值，并生成新的 streamedRowKey，如果 streamedTable

中的数据已经迭代完，则均设置为 null。

对于 bufferedTable 来说，迭代器移动得到新的 bufferedRow 由 advancedBufferedToRowWithNullFreeJoinKey 方法完成。顾名思义，该方法会跳过包含 null 的数据行，具体实现代码如下。可以看到，其迭代逻辑和 advancedStreamed 函数的迭代逻辑类似，主要区别在于如果 bufferedRowKey 中任何字段包含 null（调用 InternalRow 的 anyNull 方法），则迭代操作会继续进行，一直到得到不包含 null 的 bufferedRowKey 或 bufferedTable 数据处理完。

```
private def advancedBufferedToRowWithNullFreeJoinKey(): Boolean = {
    var foundRow: Boolean = false
    while (!foundRow && bufferedIter.advanceNext()) {
      bufferedRow = bufferedIter.getRow
      bufferedRowKey = bufferedKeyGenerator(bufferedRow)
      foundRow = !bufferedRowKey.anyNull
    }
    if (!foundRow) {
      bufferedRow = null
      bufferedRowKey = null
      false
    } else {
      true
    }
}
```

SortMergeJoinScanner 供其他模块获取数据的接口是 getStreamedRow 与 getBufferedMatches，分别用来获取当前满足 Join 条件的单个 streamedRow 和多个 bufferedRow（以数组形式缓存）。在 SortMergeJoinScanner 中，满足 Join 条件的多个 bufferedRow 存储在类型为 ArrayBuffer[InternalRow] 的 bufferedMatches 中。在 streamedTable 与 bufferedTable 两个数据表进行迭代时，如果当前 streamedRow 和 bufferedRow 能够满足 Join 条件，那么将继续移动 bufferedIter，将 bufferedTable 中满足条件的所有数据行一次性找出，存储到 bufferedMatches 中，其实现逻辑如下。可以看到，执行 bufferMatchingRows 时，仍然会再一次地检查相关的 null 情形。

```
private def bufferMatchingRows(): Unit = {
    assert(streamedRowKey != null)
    assert(!streamedRowKey.anyNull)
    assert(bufferedRowKey != null)
    assert(!bufferedRowKey.anyNull)
    assert(keyOrdering.compare(streamedRowKey, bufferedRowKey) == 0)
    matchJoinKey = streamedRowKey.copy()
    bufferedMatches.clear()
    do {
      bufferedMatches += bufferedRow.copy() // need to copy mutable rows before buffering them
      advancedBufferedToRowWithNullFreeJoinKey()
    } while (bufferedRow != null && keyOrdering.compare(streamedRowKey, bufferedRowKey) == 0)
  }
```

在 SortMergeJoinScanner 中，最重要的是 findNextInnerJoinRows 与 findNextOuterJoinRows 两个函数，它们是不断得到满足 Join 条件数据（streamedRow 与 bufferedMatches 数组）的主要驱动逻辑所在。在这两个函数中，findNextInnerJoinRows 用来得到满足 Inner Join 条件的数据，而 findNextOuterJoinRows 用来得到满足 Outer Join 条件的数据，下面详细剖析其内部实现。

如果是 Inner Join 类型，则 findNextInnerJoinRows 的逻辑相对复杂，需要将各种情形都考虑到，如 Algorithm 3 所示。

Algorithm 3 SortMergeJoinScanner.findNextInnerJoinRows

```
1:  while advancedStreamed() && streamedRowKey.anyNull do
2:    // Advance the streamed side of the join
3:  end while
4:  if streamedRow == null then
5:    matchJoinKey ← null
6:    bufferedMatches.clear()
7:    false
8:  else if matchJoinKey != null && compare(streamedRowKey, matchJoinKey) == 0 then
9:    true
10: else if bufferedRow == null then
11:   matchJoinKey ← null
12:   bufferedMatches.clear()
13:   false
14: else
15:   comp ← compare(streamedRowKey, bufferedRowKey)
16:   repeat
17:     if streamedRowKey.anyNull then
18:       advancedStreamed()
19:     else
20:       assert(!bufferedRowKey.anyNull)
21:       comp ← compare(streamedRowKey, bufferedRowKey)
22:       if comp > 0 then
23:         advancedBufferedToRowWithNullFreeJoinKey()
24:       else if comp < 0 then
25:         advancedStreamed()
26:       end if
27:     end if
28:   until streamedRow != null && bufferedRow != null && comp != 0
29:   if streamedRow == null || bufferedRow == null then
30:     matchJoinKey ← null, bufferedMatches.clear(), false
31:   else
32:     assert(comp == 0)
33:     bufferMatchingRows()
34:     true
35:   end if
36: end if
```

在执行 findNextInnerJoinRows 时，首先移动 streamedIter，得到新的 streamedRow（advancedStreamed 函数），为确保当前 streamedRowKey 中不包含 null 字段，这里执行的是 while 循环，跳过所有 streamedRowKey 包含 null 字段的数据行。在此之后，涉及 4 种情况（对应算法中的 IF 语句）。

- 如果 sreamedRow 为 null，那么就表示 streamedTable 已经处理完毕，清空 bufferedMatches 数组，返回 false 值即可。
- 如果当前的 streamedRowKey 也能够满足 bufferedMatches 中的数据，这种情况意味着和上一条数据的 streamedRowKey 相等，那么无需考察 bufferedTable 数据，直接返回 true 即可。
- 如果上述情况都不满足，且 bufferedTable 中的数据处理完毕（bufferedRow 为 null），则会清空 bufferedMatches 数组，返回 false 值。
- 最核心的部分从第 14 行开始，其逻辑与归并处理的流程相似，不断比较 streamedRowKey 和 bufferedRowKey 两个值，如果 streamedRowKey 值较小，则移动 streamedTable 的迭代器（第 25 行 advancedStreamed）；如果 bufferedRowKey 值较小，则移动 bufferedTable 的迭代器（第 23 行 advancedBufferedToRowWithNullFreeJoinKey）；如果两者相等（表示两边的数据都满足 Join 条件）或两个表中存在处理完毕的情况，则跳出循环，当能够满足 Join 条件时，执行 bufferMatchingRows 方法得到 bufferedMatches 数组。

如果是 Outer Join 类型，则 findNextOuterJoinRows 查找满足 Join 条件数据行的逻辑比较简单。实际上，Outer Join 只要 streamedTable 有数据就能够满足 Join 条件。所以，在 findNextInnerJoinRows 中只需要考虑 streamedTable 是否有数据。具体算法如 Algorithm 4 所示，在执行 findNext- InnerJoinRows 时，首先移动 streamedIter，得到新的 streamedRow（advancedStreamed 函数）。如果得到的 sreamedRow 为 null，那么就表示 streamedTable 已经处理完毕，清空 bufferedMatches 数组，返回 false 值即可；否则，考察 bufferedMatches 中的数据，如果当前的 streamedRowKey 也能够满足 bufferedMatches 中的数据，此时无需进行任何操作，直接返回 true 即可；如果 bufferedMatches 中没有数据，那么不断移动 bufferedTable 的迭代器（第 15 行 advancedBufferedToRowWithNullFreeJoinKey）直到当前 bufferedRowKey 值比 streamedRowKey 值大或两者相等（comp==0，满足 Join 条件）。当能够满足 Join 条件时，执行 bufferMatchingRows 方法得到 bufferedMatches 数组。

SortMergeFullOuterJoinScanner 是专门用于 Full Outer 类型的 Join 执行时查找匹配数据的核心类，如图 8.27 所示。不同于 SortMergeJoinScanner 中存在 streamedTable 和 bufferedTable，SortMergeFullOuterJoinScanner 中两个数据表的地位是相同的，分别称为 leftTable 和 rightTable。在其构造参数中，除左右表的迭代器（leftIter 和 rightIter）外，还有 leftNullRow 与 rightNullRow 用于原始表中数据无法匹配时填充的 null 数据行。

Algorithm 4 SortMergeJoinScanner.findNextOuterJoinRows

```
1: if !advancedStreamed() then
2:    matchJoinKey ← null
3:    bufferedMatches.clear()
4:    false
5: else
6:    if matchJoinKey != null && compare(streamedRowKey, matchJoinKey) == 0 then
7:       // Matches the current group, so do nothing
8:    else
9:       matchJoinKey ← null
10:      bufferMatchingRows()
11:      if bufferedRow != null && !streamedRowKey.anyNull then
12:         comp ← 1
13:         repeat
14:            comp ← compare(streamedRowKey, bufferedRowKey)
15:         until comp > 0 && advancedBufferedToRowWithNullFreeJoinKey()
16:      end if
17:      if comp == 0 then
18:         bufferMatchingRows()
19:      end if
20:   end if
21:   true
22: end if
```

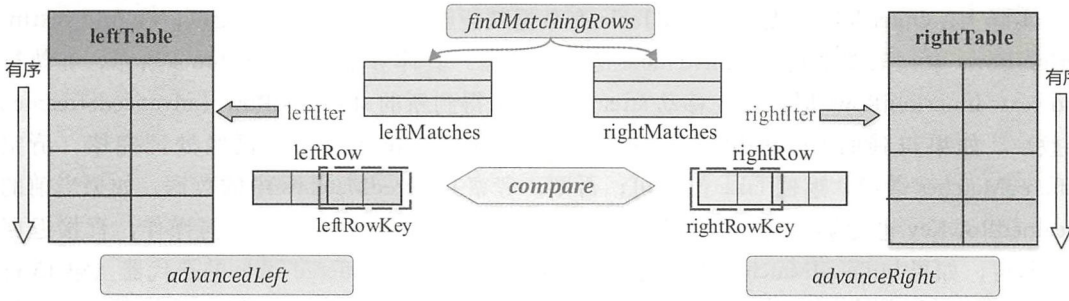


图 8.27　SortMergeFullOuterJoinScanner 实现

左表和右表分别前移的方法为 advancedLeft() 和 advancedRight()，在 SortMergeFullOuterJoinScanner 遍历数据过程中会构造两个缓冲区（leftMatches 和 rightMatches），用来缓存匹配右表当前数据行的数据与缓存匹配左表当前数据行的数据。因此，要得到 Join 之后的数据，在这两个缓冲区中查找即可，这个操作即 scanNextInBuffered() 方法。

缓冲区数据生成的主要逻辑算法如 Algorithm 5 所示，其中传入的参数 matchingKey（类型

为 InternalRow）是当前用于匹配的 key，考虑到数据是排好序的，因此不断地比较左（右）表当前数据行中的 leftRowKey（rightRowKey），如果相等，加入到 leftMatches（rightMatches）即可。

Algorithm 5 SortMergeFullOuterJoinScanner.findMatchingRows(matchingKey)

```
1: leftMatches.clear()
2: rightMatches.clear()
3: while leftRowKey != null && compare(leftRowKey, matchingKey) == 0 do
4:     leftMatches += leftRow.copy()
5:     advancedLeft()
6: end while
7: while rightRowKey != null && compare(rightRowKey, matchingKey) == 0 do
8:     rightMatches += rightRow.copy()
9:     advancedRight()
10: end while
```

8.6 本章小结

本章对 Spark SQL 中 Join 实现的内部原理进行了全面介绍。首先介绍了相关的文法定义，然后在具体实例的基础上按照每一步详细分析了如何从 Join 语句解析为逻辑计划，以及如何从逻辑计划生成物理计划，最后详细剖析 Spark SQL 中 3 种主要的 Join 实现方式，即 BroadcastJoinExec、ShuffleHashJoinExec 和 SortMergeJoinExec。

实际上，Join 的高效执行一直是各大数据库系统的“兵家必争之地”，业界也有着“得 Join 者得天下”的说法。在 Spark 2.1 版本中，Join 的实现还有着较大的优化空间，后续版本会陆续加入基于代价（Cost-based）多表优化等机制，有兴趣的读者可以继续研究。本章内容算是抛砖引玉，侧重于实现层面的剖析，读者熟悉了基本的原理后，可以针对性地进行一些改造和优化。

第9章 Tungsten 技术实现

Tungsten 是开源社区专门用于提升 Spark 性能的计划。因为 Spark 是用 Scala 语言开发的，所以最终运行在 JVM 上的代码存在着各方面的限制和弊端（例如 GC 上的 overhead），使得 Spark 在性能上有很大的提升空间。基于此考虑，Tungsten 计划应运而生，旨在从内存和 CPU 层面对 Spark 的性能进行优化。

Tungsten 的优化主要包括 3 个方面：内存管理机制（Memory management and binary processing）、缓存敏感计算（Cache-aware computation）和动态代码生成（Code generation）。官方技术文档已经对这 3 个方面的优化进行了介绍，本章将从具体的实现细节来阐述其原理。在一定程度上，这些技术具有广泛的通用性，对于目前各种分布式计算框架都有着借鉴意义。

9.1 内存管理与二进制处理

作为一个基于内存的分布式计算引擎，Spark 系统的重要特点在于多个任务之间的数据通信可以直接通过内存而避免落盘，大大提高了执行效率，因此内存管理在 Spark 系统中扮演着非常重要的角色。Tungsten 设计了一套内存管理机制，Spark 可以直接操作二进制数据而不是 JVM 对象，使得内存使用效率大幅提升。本节先详细介绍 Spark 内存管理的基础，然后分析 Tungsten 的内存管理机制，以及如何基于新的内存管理机制实现一些常用的操作。

9.1.1 Spark 内存管理基础

在分析 Tungsten 内存优化之前，有必要了解 Spark 在内存管理方面的基础原理。回顾 Spark 的计算模型，当应用程序被提交后，集群会启动 Driver 和 Executor 两种类型的服务进程，其中 Driver 为主进程，Executor 是计算单元（Task）的集合。

从本质上讲，Driver 和 Executor 都是 JVM 进程，运行在 Worker（Standalone 模式）或 Container（YARN 模式）中。因此，Spark 内存管理会涉及 Driver 和 Executor 这两种进程中内存的申请和

回收等操作。如图 9.1 所示，Driver 端会维护 Spark 的上下文环境（SparkSession/SparkContext），同时 Driver 和每个 Executor 都有自己的内存空间，内存管理则由 MemoryManager 统一管理。同一个 Executor 内的任务都会调用 MemoryManager 接口中定义的方法来完成内存申请或释放等操作。

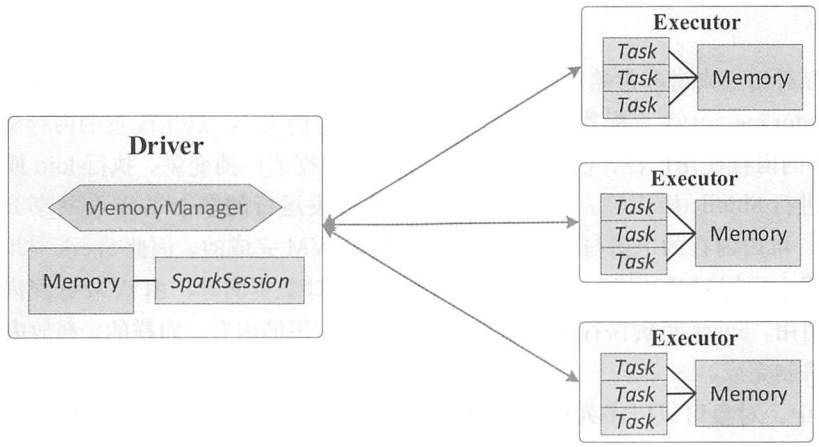


图 9.1 Spark 内存概览

Spark 为数据存储和计算执行提供了统一的内存管理接口（MemoryManager）。在具体实现上，在 Driver 端创建 MemoryManager 的过程如以下代码所示。MemoryManager 包括 StaticMemoryManager 和 UnifiedMemoryManager 两种。

```
val useLegacyMemoryManager = conf.getBoolean("spark.memory.useLegacyMode", false)
val memoryManager: MemoryManager =
    if (useLegacyMemoryManager) {
      new StaticMemoryManager(conf, numUsableCores)
    } else {
      UnifiedMemoryManager(conf, numUsableCores)
    }
```

在 Spark 1.6 版本之前，Spark 采用的是静态内存管理（StaticMemoryManager）方式。从 1.6 版本开始，Spark 默认采用统一内存管理（UnifiedMemoryManager）方式，但静态内存管理方式仍然保留（通过 spark.memory.useLegacyMode 参数配置）。相比静态内存管理方式，统一内存管理方式中存储内存和计算内存能够共享同一个空间，并且可以动态地使用彼此的空闲区域。

本节后续内容的安排如下，首先介绍 Spark 内存管理中的一些基本概念，并分析 MemoryManager 的两种实现机制（重点是 UnifiedMemoryManager），然后在此基础上展开分析存储和计算执行的内存管理实现。

1. Spark 内存管理基本概念与内存管理器 MemoryManager

（1）内存模式（MemoryMode）：Spark 系统运行在 JVM 上，因此其内存管理也建立在 JVM 的内存管理之上。MemoryManager 将内存模式分为堆内（ON_HEAP）内存和堆外（OFF_HEAP）内存。这里先简要介绍 Spark 系统中的堆内内存和堆外内存管理机制，后续内容中会详细剖析实现细节。

堆内内存对用户来讲非常熟悉，在实际 Spark 应用中用户经常使用参数（spark.driver.memory 和 spark.executor.memory）来配置 Driver 与 Executor 内存的大小。这里配置的内存就是指堆内内存。Executor 的内存使用广泛，例如，缓存 RDD 中的数据或广播变量、执行 Join 或 Aggregation 计算任务时进行 Shuffle 操作等。同一个 Executor 中并发运行的多个 Task 会共享 Executor 进程的 JVM 内存。堆内内存的申请与释放在本质上都是由 JVM 完成的，因此 Spark 对堆内内存的管理只是在逻辑上进行记录和规划。例如，Spark 在创建对象实例时，由 JVM 分配内存空间并返回该对象的引用，Spark 负责保存该引用并记录该对象占用的内存。同样的，释放内存也由 JVM 的垃圾回收机制完成。

在 JVM 中，对象可以以序列化的方式存储，序列化的过程是将对象转换为二进制字节流，在本质上可以理解为将非连续空间的链式存储转换为连续空间的块存储，在访问时需要进行序列化的逆过程——反序列化，将字节流转换为对象。序列化的方式可以节省存储空间，但增加了转换过程的计算开销。Spark 中序列化的对象是字节流的形式，其占用的内存大小可直接计算，而对于非序列化的对象，其占用的内存是通过周期性地采样近似估算而得的，并不是每次新增的数据项都会计算一次占用的内存大小，这种方法降低了时间开销，但是有可能误差较大，导致某一时刻的实际内存有可能远远超出预期。此外，被 Spark 标记为释放的对象实例，实际上很有可能并没有被 JVM 回收，导致实际可用的内存大小小于 Spark 记录的可用内存大小。所以，Spark 并不能准确记录实际可用的堆内内存，也就无法完全避免内存溢出（Out of Memory）的异常。

虽然不能精准控制堆内内存的申请和释放，但 Spark 通过对存储内存和执行内存各自独立的规划管理，可以决定是否要在存储内存里缓存新的 RDD，以及是否为新的任务分配执行内存，在一定程度上可以提升内存的利用率，减少异常的出现。

堆内内存依赖于 JVM，无法完全摆脱 GC 带来的开销，为了进一步优化内存的使用及任务执行性能，Spark 用到了堆外（OFF_HEAP）内存，使之可以直接在集群节点的系统内存中开辟空间，存储经过序列化的二进制数据。在 Spark 2.0 版本之前，堆外内存还依赖第三方的分布式内存文件系统 Tachyon[31]（后改名为 Alluxio[32]），从 2.0 版本开始，Spark 基于 JDK 自带的 Unsafe API 实现了堆外内存的管理。

JVM 堆外内存管理的引入使得 Spark 可以很方便地直接在工作节点系统内存中分配空间，能够进一步优化内存的使用，减少了不必要的内存开销（例如频繁的垃圾回收），提升了处理

性能。堆外内存可以被精确地申请和释放，而且序列化的数据占用的空间可以被精确计算。相对来讲，堆外内存比堆内内存更加容易管理。在默认情况下堆外内存并不启用，可通过参数（spark.memory.offHeap.enabled）开启，同样也可以通过参数（spark.memory.offHeap.size）控制堆外内存空间的大小。

（2）内存池（MemoryPool）：MemoryManager 通过内存池机制管理内存。简单来讲，内存池就是内存空间中一段大小可以调节的区域。在具体实现上，MemoryPool 是一个抽象类，内部定义了一个用来表示内存池大小的“_poolSize”变量和 5 个相关的函数，如图 9.2 所示。需要注意的是，MemoryPool 类的构造函数 MemoryPool（lock: Object）会通过 Object 构造参数来实现同步加锁的目的。

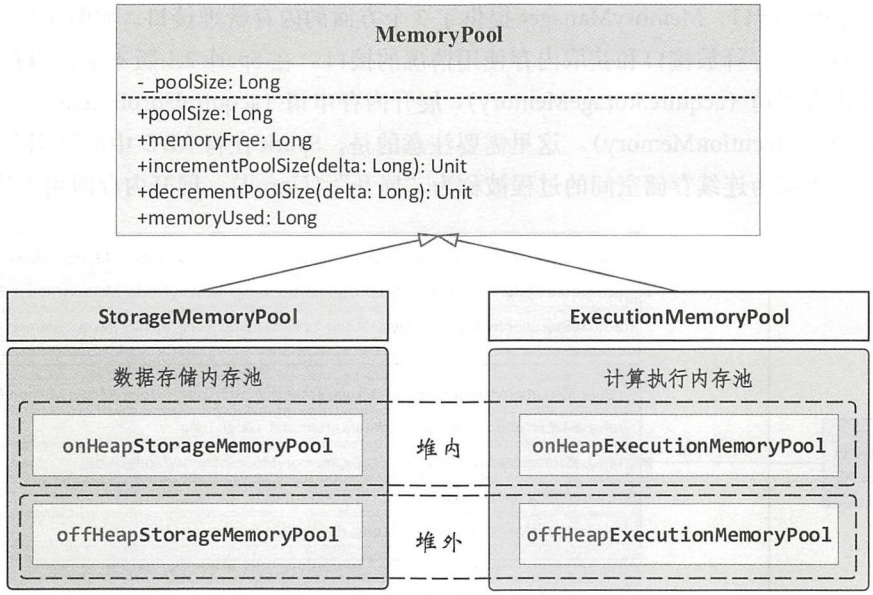


图 9.2 MemoryManager 内存池

在 Spark 中，MemoryPool 抽象类的具体实现有 StorageMemoryPool 和 ExecutionMemoryPool 两种，分别对应数据存储的内存池和计算执行的内存池。由于存在不同的内存模式（MemoryMode），MemoryManager 实际上涉及 4 个资源池。数据存储内存池包含 onHeapStorageMemoryPool（堆内）和 offHeapStorageMemoryPool（堆外）两个，计算执行内存池包含 onHeapExecutionMemoryPool（堆内）和 offHeapExecutionMemoryPool（堆外）两个。

（3）内存页（page）：MemoryManager 中设定了内存页的大小（pageSizeBytes），单位为字节。如果用户没有显示地通过配置参数（spark.buffer.pageSize）设置该值，则在 MemoryManager 中会采用以下公式来计算，从而得到默认值。

$$\text{minPageSize} = 1\text{L} * 1024 * 1024,\ \text{maxPageSize} = 64\text{L} * \text{minPageSize}$$

$$\text{safetyFactor} = 16,\ \text{size} = \text{nextPowerOf2}\left(\frac{\text{maxTungstenMemory}}{\text{cores} * \text{safetyFactor}}\right)$$

$$\text{default} = \text{math.min}(\text{maxPageSize}, \text{math.max}(\text{minPageSize}, \text{size}))$$

公式中的 maxTungstenMemory 为计算执行内存池的大小，这个值为 onHeapExecutionMemoryPool 还是 offHeapExecutionMemoryPool 的内存池大小取决于 tungstenMemoryMode 对应的内存模式。变量 cores 为 CPU 的核数，如果 MemoryManager 中没有设置，会直接调用操作系统接口得到。此外，nextPowerOf2(n) 函数得到不小于 n 且为 2 的幂的最小值，例如，nextPowerOf2(3)=4。那么根据公式可知，最小的内存页大小为 1MB，最大的内存页大小为 64MB。

(4) 内存管理接口：MemoryManager 提供了 3 个方面的内存管理接口，如图 9.3 所示，包括内存申请接口、内存释放接口和获取内存使用情况的接口。在 Spark 2.1 版本中，内存申请接口又包含存储内存申请（acquireStorageMemory）、展开内存申请（acquireUnrollMemory）和执行内存申请（acquireExecutionMemory）。这里需要注意的是，Spark 在将 RDD 中的数据分区由不连续的存储空间组织为连续存储空间的过程被称为“展开”（Unroll），展开内存即用于这个操作。

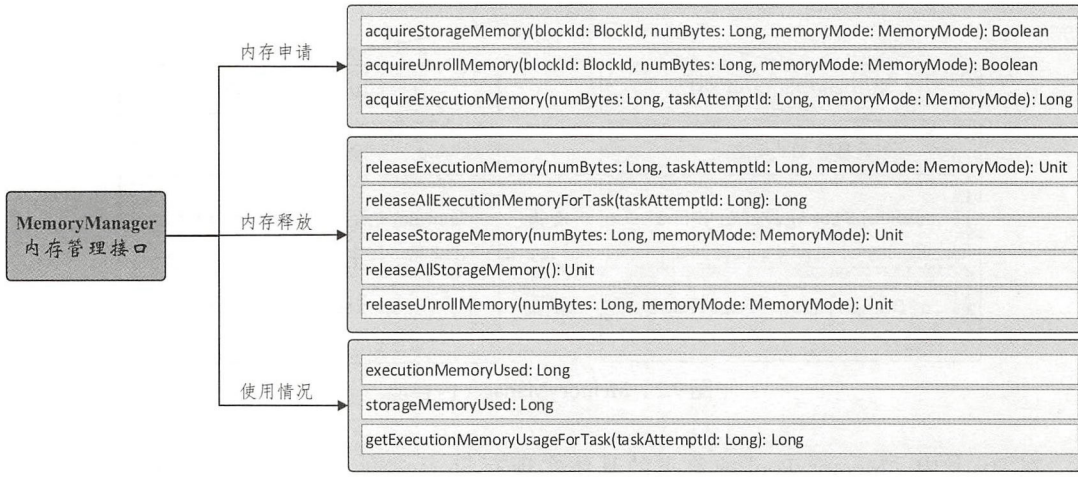


图 9.3　MemoryManager 内存管理接口

(5) 静态内存管理机制（StaticMemoryManager）：StaticMemoryManager 是 1.6 版本之前唯一的内存管理器，当前版本中都默认设置为关闭状态。将其称为静态内存管理的主要原因在于数据存储与计算执行的内存占比和界限是固定的，彼此不能互相占用内存。从实现上来看，StaticMemoryManager 的代码非常简单，主要重载实现了抽象类中的 acquireStorageMemory、acquireUnrollMemory 和 acquireExecutionMemory 3 个内存申请方法。需要注意的是，静态内存管理器中是不支持堆外内存作为数据存储的，因此实际上只有 3 个内存池，并且在申请数据存

储内存时（acquireStorageMemory），内存模式（MemoryMode）应该是堆内模式（ON_HEAP）。

总体来看，MemoryManager 会将内存分为存储内存、执行内存和其他 3 部分，如图 9.4 所示。数据存储内存的比例通过参数 spark.storage.memoryFraction 设置，默认比例为 0.6，Unroll 部分内存也属于存储内存的一部分；计算执行内存的比例通过参数 spark.shuffle.memoryFraction 设置，默认比例为 0.2；剩下的部分内存被系统保留，通常用来存储运行中产生的一些对象。

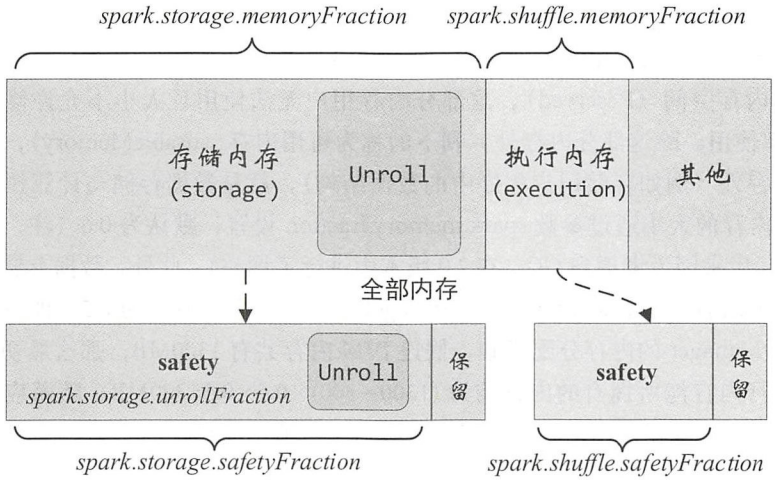


图 9.4 StaticMemoryManager 内存分配

通过上述比例得到的数据存储内存和计算执行内存一般不会被完全使用，为了防止内存溢出，都会按照一定比例设置安全的使用空间。如图 9.4 所示，计算执行内存部分的安全比例通过参数 spark.shuffle.safetyFraction 设置，默认为 0.8；数据存储内存部分的安全比例通过参数 spark.storage.safetyFraction 设置，默认为 0.9，Unroll 内存处于安全空间中，同样可以通过参数 spark.storage.unrollFraction 设置，默认比例为 0.2。例如，如果 JVM 的内存共 100GB，则在默认参数设置下，系统保留 100×0.2 = 20GB；计算执行内存共 100×0.2 = 20GB，其中安全空间为 20×0.8 = 16GB，保留 4GB；数据存储内存共 100×0.6 = 60GB，其中保留 60×0.1 = 6GB，安全空间为 60×0.9 = 54GB，在安全空间中，Unroll 内存为 54×0.2 = 10.8GB。

（6）统一内存管理机制（UnifiedMemoryManager）：统一内存管理机制是从 1.6 版本开始实现的内存管理器，其设计思路可以参见相关文档[33]。统一内存管理机制旨在解决静态内存管理方式中不断凸显的局限性。

- 静态的方式往往无法高效地适用于所有的应用类型，例如，大量 Cache 数据的应用与大量 Join 操作的应用对于存储和执行的内存需求必然不一样。
- 静态内存管理方式一般都需要调整相关参数，对于用户的 Spark 内核知识要求很高。

- 静态的方式也容易造成内存资源的浪费，例如计算执行需要大量内存，而存储内存却空着无法使用。

从实现上来看，UnifiedMemoryManager 同样也是 MemoryManager 的子类，重载实现了 acquireStorageMemory、acquireUnrollMemory 和 acquireExecutionMemory 3 个内存申请的方法。需要注意的是，UnifiedMemoryManager 中不再单独管理 Unroll 内存的申请，而是统一到数据存储内存的管理中。

UnifiedMemoryManager 的内存分配机制如图 9.5 所示，比较明显的是其中多了一个 300MB 的固定保留的内存空间（Reserved），这部分内存用户无法使用且大小不允许修改，一般用于 Spark 系统内部使用。除这部分内存外，剩下的称为可用内存（usableMemory），可用内存中除留给应用的内存外（例如存储用户程序中的数据结构），就是数据存储与计算执行公用的内存区域。这部分内存的大小通过参数 spark.memory.fraction 设置，默认为 0.6（注：设计文档中初始实现为 0.75，后来因为考虑到 GC，在 2.0 版本中进行了调整）。此外，数据存储内存空间大小的初始比例也可以通过参数 spark.memory.storageFraction 设置，默认为 0.5，即各占一半。根据 UnifiedMemoryManager 的内存分配机制，假设 JVM 内存共有 1300MB，那么最初状态下，数据存储和计算执行内存池所拥有的内存均为 (1300−300)×0.6×0.5=300MB。随着应用的执行，数据存储与计算执行间互相借用内存，可能会导致这个数值不断变化。

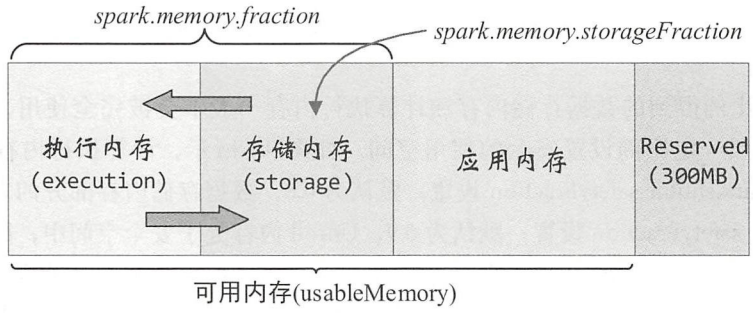


图 9.5 UnifiedMemoryManager 的内存分配机制

统一内存管理最大的特点在于动态占用机制，其规则如下。

- 设定基本的存储内存和执行内存区域（使用 spark.storage.storageFraction 参数），该设定确定了双方各自拥有的空间范围。
- 双方的空间都不足时，则存储到硬盘，若己方空间不足而对方空间空余时，可借用对方的空间（注：存储空间不足是指不足以放下一个完整的 Block）。
- 执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后“归还”借用的空间。

- 存储内存的空间被对方占用后，无法让对方“归还”，因为需要考虑 Shuffle 过程中的很多因素，实现起来较为复杂。

基于上述统一内存管理机制，Spark 系统能够有效地提高堆内内存和堆外内存资源的利用率，避免了静态内存管理方式中烦琐的配置维护操作。然而，统一内存管理并不意味着一劳永逸，例如某些场景下如果缓存的数据过多，反而会导致 JVM 频繁地进行全量垃圾回收（Full GC），影响任务执行的性能。

2. 存储内存管理

存储内存管理的入口是 StorageMemoryPool，如图 9.6 所示，StorageMemoryPool 是 MemoryPool 的子类，构造参数是用于加锁的对象（Lock）和内存模式（MemoryMode）。StorageMemoryPool 内部包括 poolName、_memoryUsed 和 _memoryStore 3 个变量。其中，poolName 是内存池的名称（on-heap execution 或 off-heap execution）；_memoryUsed 用于记录当前存储内存池的内存使用量；_memoryStore 是 MemoryStore 类型，通过 setMemoryStore 方法设置，可以看作是 Spark 存储内存管理的接口。

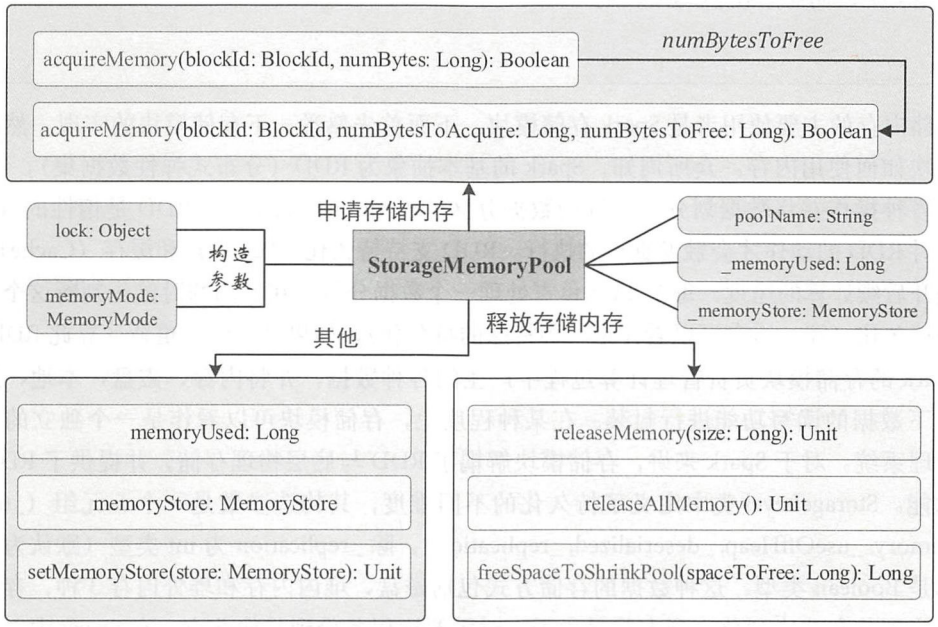


图 9.6 数据存储内存池（StorageMemoryPool）

StorageMemoryPool 的实现比较简单，图 9.6 中比较重要的是用来申请内存的 acquireMemory 方法。跟踪其源码可知，acquireMemory 方法最终调用的是 MemoryStore 中的 evict-

BlocksToFreeSpace 方法，将内存中的数据块落盘，回收相应的内存空间。如果 MemoryStore 回收了数据块的内存空间（“Evict blocks”），则同步机制会动态更新 StorageMemoryPool 中的内存使用量，相应的变量也会自动完成更新，不需要调用程序显示地修改。此外，StorageMemoryPool 中也提供了释放部分内存的 releaseMemory 方法和释放全部存储内存的 releaseAllMemory 方法。

当需要给计算执行层“借”内存时，StorageMemoryPool 中的 freeSpaceToShrinkPool 方法提供了“收缩（Shrink）”存储内存区可用内存来支持。具体实现如以下代码所示，该方法最终仍然调用 MemoryStore 中的 evictBlocksToFreeSpace 方法。需要注意的是，freeSpaceToShrinkPool 方法调用后不会更新当前 StorageMemoryPool 的大小，需要调用的程序显示地应用 decrementPoolSize 方法来处理。

```
def freeSpaceToShrinkPool(spaceToFree: Long): Long = lock.synchronized {
    val spaceFreedByReleasingUnusedMemory = math.min(spaceToFree, memoryFree)
    val remainingSpaceToFree = spaceToFree - spaceFreedByReleasingUnusedMemory
    if (remainingSpaceToFree > 0) {
      val spaceFreedByEviction =
        memoryStore.evictBlocksToFreeSpace(None, remainingSpaceToFree, memoryMode)
      spaceFreedByReleasingUnusedMemory + spaceFreedByEviction
    } else {
      spaceFreedByReleasingUnusedMemory
    }
}
```

存储内存的主要使用者是 Spark 存储模块。下面首先熟悉一下存储模块的实现，然后探讨存储模块如何使用内存。众所周知，Spark 的基本抽象为 RDD（分布式弹性数据集），RDD 中定义了各种操作并将数据划分为只读的数据分区（Partition）的集合。RDD 是惰性的（Lazy），只有针对 RDD 的动作才会触发真正的执行。RDD 支持持久化（Persist）和缓存（Cache）操作，以便提升后续计算的速度。每个 Task 负责处理一个数据分区，在启动的时候先判断这个分区是否已经持久化（注：缓存可以看作是一种特殊的持久化），如果没有则会重新计算此 RDD。

Spark 的存储模块负责管理计算过程中产生的各种数据，并将内存、磁盘、本地、远程各种场景下数据的读写功能进行封装。在某种程度上，存储模块可以看作是一个独立的分布式存储管理系统。对于 Spark 来讲，存储模块解耦了 RDD 与底层物理存储，并提供了 RDD 的持久化功能。StorageLevel 类中定义了持久化的不同维度，其构造参数是一个五元组（_useDisk, _useMemory, _useOffHeap, _deserialized, _replication），除 _replication 为 int 类型（默认为 1）外，其他都是 Boolean 类型。这种数据的存储方式包括磁盘、堆内内存和堆外内存 3 种，存储形式可以是序列化和非序列化，副本数量大于 1 时需要远程备份到其他节点。在 Spark 中，不同的 StorageLevel 方式如表 9.1 所示，例如 MEMORY_AND_DISK 意味着同时在磁盘和堆内内存上存储数据。

具体到实现层面，存储模块的入口为 BlockManager 类，整体架构采用的是主从模式（注：

表 9.1 不同的 StorageLevel 方式

StorageLevel 方式	构造参数
NONE	StorageLevel(false, false, false, false)
DISK_ONLY	StorageLevel(true, false, false, false)
DISK_ONLY_2	StorageLevel(true, false, false, false, 2)
MEMORY_ONLY	StorageLevel(false, true, false, true)
MEMORY_ONLY_2	StorageLevel(false, true, false, true, 2)
MEMORY_ONLY_SER	StorageLevel(false, true, false, false)
MEMORY_ONLY_SER_2	StorageLevel(false, true, false, false, 2)
MEMORY_AND_DISK	StorageLevel(true, true, false, true)
MEMORY_AND_DISK_2	StorageLevel(true, true, false, true, 2)
MEMORY_AND_DISK_SER	StorageLevel(true, true, false, false)
MEMORY_AND_DISK_SER_2	StorageLevel(true, true, false, false, 2)
OFF_HEAP	StorageLevel(true, true, true, false, 1)

Driver 端的 BlockManager 为主，Executor 端的 BlockManager 为从）。数据存储的基本单位为数据块（Block），每个 Block 都会产生唯一的 BlockId 标识。RDD 本身、Shuffle、Broadcast 等过程都会涉及数据存储，因此也会产生对应的 BlockId，例如 RDD 中每个分区（Partition）唯一对应一个 Block，此时 BlockId 的格式为“rdd_rddId_partitionId”。应用执行过程中，Driver 端的 BlockManager 负责对全部数据块（Block）的元数据信息进行管理和维护，Executor 端的 BlockManager 将数据块（Block）的更新等状态上报到 Driver 端，并接收主节点的相关操作命令。

Spark 的 RDD 在缓存到存储内存之前，每条数据的对象实例都处于 JVM 堆内内存的 Other 部分，即便同一个分区内的数据在内存空间中也是不连续的，具体分布由 JVM 管理，上层通过 Scala 中的迭代器来访问。当 RDD 持久化到存储内存之后，Partition 对应转换为 Block，此时数据在存储内存空间（堆内或堆外）中将连续的存储。这里将 Partition 由不连续的存储空间转换为连续的存储空间的过程，就是前面所提到的“展开（Unroll）”操作。

根据定义的存储级别（StorageLevel），Block 也有序列化和非序列化两种存储格式。非序列化的 Block 定义在 DeserializedMemoryEntry 类中，用一个数组存储所有的对象实例；序列化的 Block 则以 SerializedMemoryEntry 类来定义，用字节缓冲区（ByteBuffer）存储二进制数据。每个 Executor 的存储模块采用 LinkedHashMap 数据结构来管理堆内和堆外存储内存中所有的 Block 对象的实例，对该 LinkedHashMap 新增和删除间接记录了内存的申请和释放。

因为不能保证存储空间可以一次容纳 Iterator 中所有的数据，所以当前的计算任务在执行“展开”操作时需要向 MemoryManager 申请足够的空间来临时占位，空间不足则展开失败，空间足够时可以继续进行。对于序列化的 Partition，其所需的展开空间可以直接累加计算，一次申请；而非序列化的 Partition 则要在遍历数据的过程中依次申请，即每读取一条 Record，采样估算其所需的展开空间并进行申请，空间不足时可以中断，释放已占用的展开空间。如果最终展开成功，则当前 Partition 所占用的展开空间被转换为正常的缓存 RDD 的存储空间。

同一个 Executor 的所有计算任务共享有限的存储内存空间，当有新的 Block 需要缓存但是剩余空间不足且无法动态占用时，就要对 LinkedHashMap 中的旧 Block 进行淘汰（Eviction）。对于被淘汰的 Block，如果其存储级别中同时包含存储到磁盘的要求，则要对其进行落盘，否则直接删除该 Block。存储内存的淘汰规则如下。

- 被淘汰的旧 Block 要与新 Block 的内存模式（MemoryMode）相同，即同属于堆外或堆内内存。
- 新旧 Block 不能属于同一个 RDD，避免循环淘汰。
- 旧 Block 所属 RDD 不能处于被读状态，避免引发一致性问题。
- 遍历 LinkedHashMap 中的 Block，按照最近最少使用（LRU）的顺序淘汰，直到满足新 Block 所需的空间。其中，LRU 是 LinkedHashMap 的特性。落盘的流程比较简单，如果其存储级别符合 useDisk 为 true 的条件，就根据其 deserialized 判断是否是非序列化的形式，若是则对其进行序列化，最后将数据存储到磁盘，在存储模块中更新其信息。

3. 执行内存管理

执行内存管理的入口是 ExecutionMemoryPool，如图 9.7 所示。ExecutionMemoryPool 继承自 MemoryPool，构造参数是用于加锁的对象（Lock）和内存模式（MemoryMode）。ExecutionMemoryPool 内部包括 poolName 和 memoryForTask 两个变量，其中 poolName 是内存池的名称（on-heap execution 或 off-heap execution），memoryForTask 实际上是一个 HashMap 结构，记录了每个 Task 的内存使用量。

ExecutionMemoryPool 中比较核心的操作是 acquireMemory 方法，numBytes 和 taskAttemptId 分别表示申请的内存大小（字节数目）和发出内存申请的 Task。需要注意的是剩下的两个参数，maybeGrowPool 是一个回调函数，用来控制增加当前 ExecutionMemoryPool 的大小（从 StorageMemoryPool 中“借”内存）；computeMaxPoolSize 同样是一个回调函数，用来获取当前 ExecutionMemoryPool 的大小。这两个函数都在 UnifiedMemoryManager 内部定义。

Executor 内运行的任务同样共享执行内存。假设当前 Executor 中正在执行的任务数目为 n，那么每个任务可占用的执行内存大小的范围为 $[1/2n, 1/n]$。每个任务在启动时，要向 MemoryManager 申请最少 $1/2n$ 的执行内存，如果不能满足要求，则该任务被阻塞，直到有其他任务释放了足够的执行内存，该任务才能被唤醒。在执行期间，Executor 中活跃的任务数目是不断变化的，Spark 采用 wait 和 notifyAll 机制同步状态并重新计算 n 的值。

执行内存主要用于满足 Shuffle、Join、Sort、Aggregation 等计算过程中对内存的需求，其中 Shuffle 算是各分布式系统中通用的底层操作，服务于上层的各种复杂计算。作为基础，本节主要介绍 Shuffle 过程的实现和该过程中内存的使用。

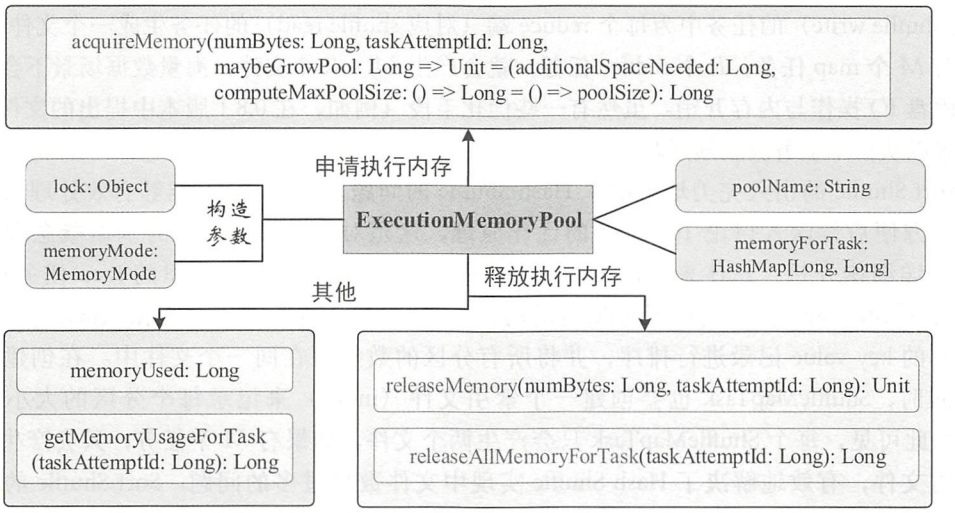


图 9.7 计算执行内存池（ExecutionMemoryPool）

首先回顾一下 Spark 中 Shuffle 实现方式的演化历程，如图 9.8 所示，经历了 Hash、Sort 和 Tungsten-Sort 3 个重要的阶段。

- 在 1.1 之前的版本中，简单采用基于 Hash 方式的 Shuffle 实现，从 1.1 版本开始 Spark 参考 MapReduce 的实现，引入 Sort Shuffle 机制（设计文档可参考 SPARK-2045 [34]），自此 Hash Shuffle 与 Sort Shuffle 共同存在，并在 1.2 版本时将 Sort Shuffle 设置为默认的 Shuffle 方式。
- 发展到 1.4 版本时，Tungsten 引入了 UnSafe Shuffle 机制来优化内存和 CPU 的使用 [35]，并在 1.6 版本中进行代码优化，统一实现到 Sort Shuffle 中，通过 Shuffle 管理器自动选择最佳 Shuffle 方式（SPARK-14667 [36]）。
- 到 2.0 版本时，Hash Shuffle 的实现从 Spark 中删除了 [37]，所有 Shuffle 方式全部统一到 Sort Shuffle 一个实现中。

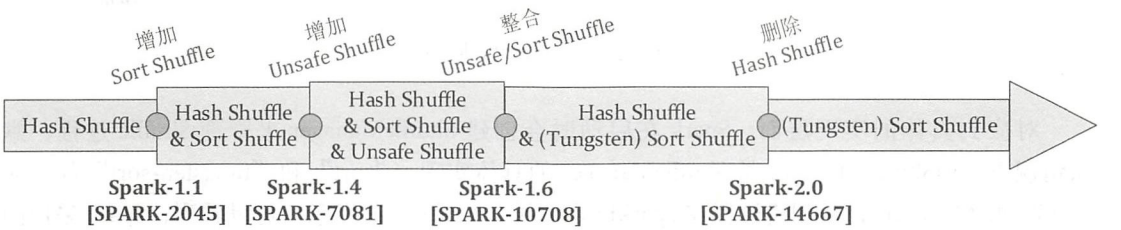


图 9.8 Spark Shuffle 实现方式的演化历程

关于 Hash 方式的 Shuffle 实现，顾名思义，其主要思想是按照 Hash 的方式在每个 map 端

(对应 shuffle write) 的任务中为每个 reduce 端 (对应 shuffle read) 的任务生成一个文件。因此，如果有 M 个 map 任务，R 个 reduce 任务，就会产生 $M \times R$ 个文件，海量数据场景下会导致大量的磁盘 IO 操作与内存开销。虽然有一些优化手段 (例如，在 0.8.1 版本中提出的文件合并)，但是难以从根本上解决瓶颈问题。

Sort Shuffle 的引入完美地解决了 Hash Shuffle 的问题，使得 Spark 能够有效处理大规模数据。为方便后续深入讨论 Tungsten 的优化原理，这里对 Sort Shuffle 中的基本概念和实现原理进行详细体分析。总体来看，Sort Shuffle 的原理如图 9.9 所示，这里的 map 任务在 Spark 中称为 ShuffleMapTask，负责 Shuffle 写的动作。在 ShuffleMapTask 中会根据 (partition id, key) 对所有的 key-value 记录进行排序，并将所有分区的数据写在同一个文件中。在创建数据文件的同时，ShuffleMapTask 也会创建一个索引文件 (index) 来记录每个分区的大小和偏移量。由此可见，每个 ShuffleMapTask 只会产生两个文件，如果有 M 个任务，只会产生 $2 \times M$ 个临时文件，有效地解决了 Hash Shuffle 实现中文件数目过多的问题。Sort Shuffle 的入口是 SortShuffleManager，ShuffleMapTask 会从 SortShuffleManager 中得到 ShuffleWriter 对象来执行写操作，需要注意的是写操作会将数据写在本地目录中；对应的，在 Shuffle 执行读任务时，每个 Task 会从 SortShuffleManager 中获得 ShuffleReader 对象，由该对象到特定的服务器节点上读取特定的数据。

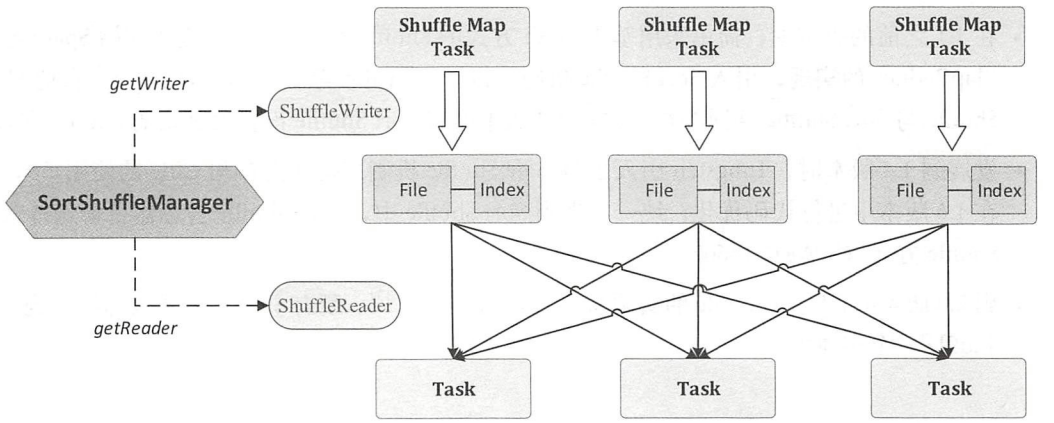


图 9.9　Sort Shuffle 原理

对应到具体的实现层面，Spark 在启动时会创建 ShuffleManager 来管理 Shuffle 过程，默认情况下 SortShuffleManager 是 ShuffleManager 的具体实现 (“sort” 和 “tungsten-sort” 都对应 SortShuffleManager)，其创建过程在 SparkEnv 类中，如以下代码所示。可以看到，Spark 采用的是一种插件式的 Shuffle 模块管理方式，开发人员可以实现自己的 ShuffleManager，并修改对应的配置参数 (spark.shuffle.manager)。

```
val shortShuffleMgrNames = Map(
    "sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].getName,
    "tungsten-sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].getName)
val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")
val shuffleMgrClass = shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase, shuffleMgrName)
val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)
```

SparkEnv 类中创建 ShuffleManager 采用的是单例模式，因此 Shuffle 读和写的任务中都是通过单例访问方式直接得到（SparkEnv.get.shuffleManager）对象的。有了 ShuffleManager 之后，从中获取 ShuffleReader 和 ShuffleWriter 对象来执行读和写操作。因此，Spark 中 Shuffle 模块的实现就比较清晰了，主要的类如图 9.10 所示。ShuffleManager、ShuffleWriter 和 ShuffleReader 都是 trait 类型，其中定义的方法也非常容易理解。ShuffleReader 只有 BlockStoreShuffleReader 一种实现，ShuffleWriter 有 BypassMergeSortShuffleWriter、SortShuffleWriter 和 UnsafeShuffleWriter 3 种实现方式，从 ShuffleManager 获取 ShuffleWriter 时会根据相关信息自动选择一种。

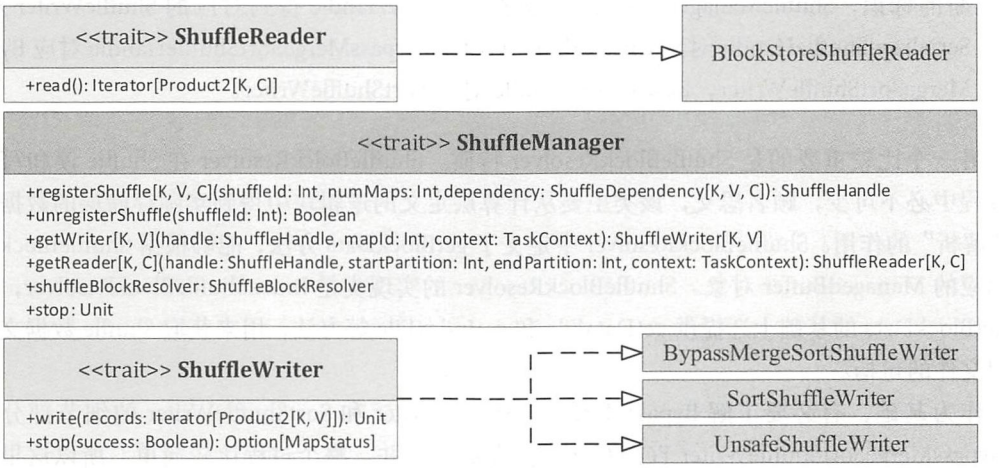


图 9.10 Shuffle 模块的实现

在进入核心逻辑分析之前，先了解 Shuffle 模块实现中的一些辅助类。首先是 ShuffleHandle 类，当 ShuffleDependency 注册一个 Shuffle 时就会得到一个 ShuffleHandle 对象，该对象中保存了 shuffleId、ShuffleMapTask 的个数和 ShuffleDependency 对象本身。在目前的 Spark 实现中，共有 3 种 ShuffleHandle。

- BypassMergeSortShuffleHandle，即可以忽略掉聚合排序的 Shuffle 过程，本质上和 Hash Shuffle 机制类似。如果当前的依赖不需要进行 map 端的聚合，且 Partitioner 分区数目小于或等于阈值（可设置 spark.shuffle.sort.bypassMergeThreshold，默认为 200），那么注册该 Shuffle 就会得到 BypassMergeSortShuffleHandle 对象，这种情况下 Shuffle 写操作时内部不

使用 MergeSort 方式处理数据，而是直接将每个分区写入单独的文件，并在最后做一个合并处理，并创建一个 index 索引文件来标记不同分区的位置信息。但是从 Shuffle 数据读取任务看来，数据文件和索引文件的格式和内部是否做过聚合排序是完全相同的。这个可以看作 Sort Based Shuffle 在 Shuffle 量比较小的时候对 Hash Based Shuffle 的一种折衷，也存在同时打开文件过多导致内存占用增加的问题。

- SerializedShuffleHandle，对应 Tungsten 方式的 Shuffle 过程，这种情况下 ShuffleMapTask 的输出数据能够先序列化为二进制数据存储在内存中，再执行相关的操作，在内存使用上是一种更高效的方式。采用 Tungsten 方式需要满足 3 个条件，即序列化框架 Serializer 支持对象重定位（Supports Relocation Of Serialized Objects）、依赖中没有定义聚合操作和分区数目不能超过阈值（16777216）。
- BaseShuffleHandle，在不满足上述要求的情况下，返回的是 BaseShuffleHandle 对象，意味着以反序列化的格式处理 Shuffle 输出数据。因此，实际上 ShuffleHandle 相当于判断的标识，ShuffleManager 根据具体是哪种 ShuffleHandle 得到对应的 ShuffleWriter 对象，SerializedShuffleHandle 对应 UnsafeShuffleWriter，BypassMergeSortShuffleHandle 对应 BypassMergeSortShuffleWriter，BaseShuffleHandle 对应 SortShuffleWriter。

另一个比较重要的是 ShuffleBlockResolver 特质。ShuffleBolckResolver 在 Shuffle 读和写数据的过程中必不可少，顾名思义，该类主要从计算层定义的逻辑块中得到实际存储层的数据，起到“解析”的作用。ShuffleBlockResolver 中定义了 getBlockData 方法，能够根据 ShuffleBlockId 获得相应的 ManagedBuffer 对象。ShuffleBlockResolver 的实现类是 IndexShuffleBlockResolver，该类在 getBlockData 的基础上还提供 getDataFile 和 getIndexFile 等方法，用来获取 Shuffle 数据文件与索引文件的句柄。

作为基础，有必要了解 BypassMergeSortShuffleWriter 和 SortShuffleWriter 的细节部分，其中 BypassMergeSortShuffleWriter 直接写文件并在最后合并，整个过程比较简单。所以这里仅对 SortShuffleWriter 的实现机制进行分析，Tungsten 优化实现的 UnsafeShuffleWriter 会在后续内容中进行介绍。SortShuffleWriter 的 write 方法实现的核心逻辑如图 9.11 中的代码所示。其主要过程非常清晰，分为 3 步。

（1）创建 ExternalSorter 对象，将全部数据插入到该对象中。

（2）生成 Shuffle 数据文件和索引文件。

（3）创建 MapStatus 对象，将 Shuffle 数据文件和索引文件的信息进行传输，这样 Shuffle 数据读取阶段能够知道从哪些节点获取哪些数据。

由此可知，SortShuffleWriter 执行 Shuffle 写操作的最终实现都落在了 ExternalSorter 类上，对于用户来讲，可以将 ExternalSorter 看作一个黑盒，对外主要提供 insertAll 和 writePartitionedFile 两个接口方法。

```
override def write(records: Iterator[Product2[K, V]]): Unit = {
  sorter = if (dep.mapSideCombine) {
    new ExternalSorter[K, V, C](context, dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.serializer)
  } else {
    new ExternalSorter[K, V, V](context, aggregator = None, Some(dep.partitioner), ordering = None, dep.serializer)
  }
  sorter.insertAll(records)                                    ①构造ExternalSorter，插入数据

  val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)
  val tmp = Utils.tempFileWith(output)                         ②生成数据文件和索引文件
  val blockId = ShuffleBlockId(dep.shuffleId, mapId, IndexShuffleBlockResolver.NOOP_REDUCE_ID)
  val partitionLengths = sorter.writePartitionedFile(blockId, tmp)
  shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId, partitionLengths, tmp)

  mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)   ③创建MapStatus，传递Shuffle信息
}
```

图 9.11 SortShuffleWriter 实现逻辑

ExternalSorter 属于 Spark 的 util 模块中的工具类，用来完成 SortMerge 过程，能够将（K, V）键值对数据排序，且支持按照 K 来聚合并得到（K, C）。ExternalSorter 主要实现逻辑是根据 Partitioner 将数据划分为不同的分区，每个分区内采用定制化的比较器（Comparator）根据 K 进行排序。ExternalSorter 具体实现原理如图 9.12 所示，构造参数包括 Task 上下文信息(TaskContext)、聚合操作类（Aggregator）、分区操作类（Partitioner）、排序操作类（Ordering）和序列化框架（Serializer）。其中，Aggregator、Partitioner 和 Ordering 为 Option 类型（虚线表示）。ExternalSorter 中创建了两种核心数据结构：一种是 PartitionedPairBuffer，本质上是内存数组，当不需要进行 map 端聚合操作时会用到；另一种是 PartitionedAppendOnlyMap，根据其命名可以知道是一种仅支持数据追加的 Map 数据结构，最终调用的是 AppendOnlyMap 数据结构，其底层实现类似 Java 中的 HashMap，基于二次再散列算法（Quadratic Probing）[38]，本质上仍然是内存数组。

ExternalSorter 插入数据的接口是 insertAll 方法，实现层面会根据 map 端数据是否需要聚合分为两种情况。第一种是没有定义 Aggregator，map 阶段不需要进行数据聚合（Combine）操作，此时数据会存储到 PartitionedPairBuffer 中。跟踪其实现源码可知，PartitionedPairBuffer 是一种可以支持动态扩容的数组，初始分配的大小为 2×64=128，可以存储 64 组（K, V）键值对。随着数据的增加，每次扩容都是当前容量的 2 倍，最多支持存储 1073741823 个键值对。从图 9.12的代码实现中可以看到，PartitionedPairBuffer 中插入的是一个三元组（partitionId, K, V），这里的 partitionId 是 Partitioner 根据 K 得到的数据所在的分区 ID，这样最终实际存储的数据是（partitionId, K）和 V，占据 PartitionedPairBuffer 中两个位置。假设（partitionId, K）的下标为 2×idx，那么 V 的下标就是 2×idx+1，这样可以保证数据相邻存储。

从代码中可以看到，每次插入一条数据后都会调用 maybeSpillCollection 检查内存的使用量，

判断是否需要将 PartitionedPairBuffer 中的数据落到磁盘。maybeSpillCollection 的实现参见如下代码，该方法最终会调用 Spillable 中定义的 maybeSpill 方法完成落盘操作，然后重新创建一个新的 PartitionedPairBuffer 对象。Spillable 中最终的判断逻辑是这样的：假设当前存储在内存中的记录总数为 elementsRead，currentMemory 是对 Buffer 中的总记录数据大小（字节数）的估算，myMemoryThreshold 通过配置项“spark.shuffle.spill.initialMemoryThreshold”进行设置，默认值为 5×1024×1024=5MB。当满足条件 elementsRead%32==0&&currentMemory >=myMemory-Threshold）时，会先尝试向 MemoryManager 申请 2×currentMemorymyMemoryThreshold 大小的内存。如果能够申请到所需要的内存空间，则不进行落盘操作，而是继续向 Buffer 中存储数据；如果申请不到内存，则调用 spill() 方法将 Buffer 中存储的数据序列化到磁盘文件。

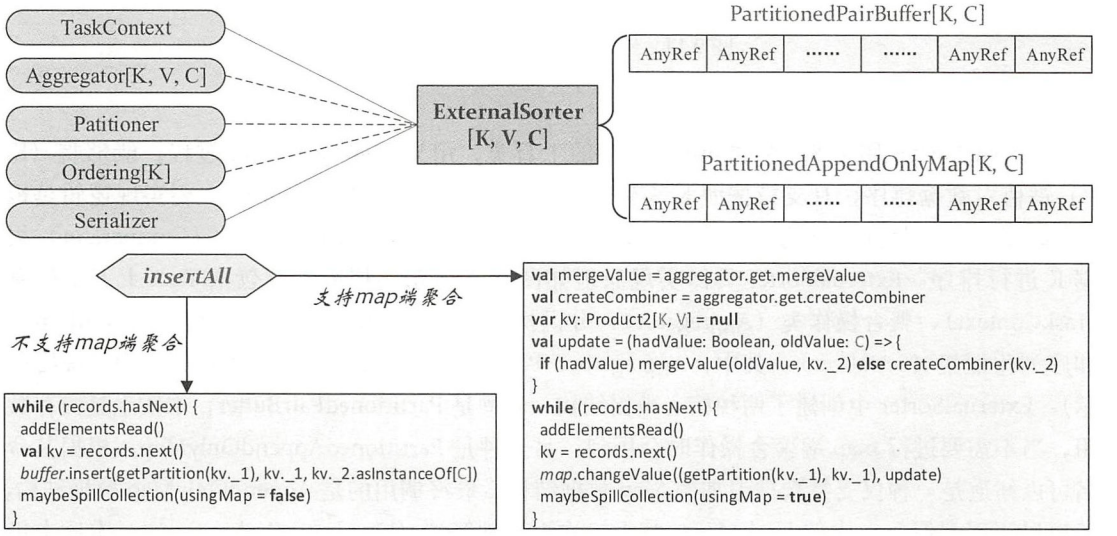


图 9.12　ExternalSorter 实现原理

```
private def maybeSpillCollection(usingMap: Boolean): Unit = {
    var estimatedSize = 0L
    if (usingMap) {
      estimatedSize = map.estimateSize()
      if (maybeSpill(map, estimatedSize)) {
        map = new PartitionedAppendOnlyMap[K, C]
      }
    } else {
      estimatedSize = buffer.estimateSize()
      if (maybeSpill(buffer, estimatedSize)) {
        buffer = new PartitionedPairBuffer[K, C]
      }
    }
    if (estimatedSize > _peakMemoryUsedBytes) {
```

```
      _peakMemoryUsedBytes = estimatedSize
    }
  }
```

当 map 端需要进行聚合时会执行 Combine 操作，在 map 阶段进行 Combine 操作能够有效降低 map 阶段数据记录的总数，从而降低 Shuffle 过程中数据的跨网络复制传输。这时，RDD 对应的 ShuffleDependency 需要设置一个 Aggregator 来执行 Combine 操作。在图 9.12的代码中，map 是内存数据结构，最重要的是 update 函数和 map 的 changeValue 方法（这里的 map 对应的实现类是 PartitionedAppendOnlyMap）。update 函数所做的工作其实就是对 createCombiner 和 mergeValue 两个函数的使用，第一次遇到一个 Key 时调用 createCombiner 函数进行处理，非首次遇到同一个 Key 对应新的 Value 时调用 mergeValue 函数进行合并处理。map 的 changeValue 方法主要是将 Key 和 Value 在 map 中存储或进行修改（对出现的同一个 Key 的多个 Value 进行合并，并用合并后的新 Value 替换旧 Value）。PartitionedAppendOnlyMap 是一个经过优化的哈希表，它支持向 map 中追加数据，以及修改 Key 对应的 Value，但是不支持删除某个 Key 及其对应的 Value。它能够支持的存储容量是 $0.7 \times 2^{29} = 375809638$。当达到指定存储容量或指定限制时，就会将 map 中的记录数据溢存到磁盘文件，这个过程和前面类似，不再赘述。

最后，在 Shuffle 数据读阶段，BlockStoreShuffleReader 实现逻辑如图 9.13 所示。读取远端节点或本地读取中间结果的处理步骤如下。

（1）获取 ShuffleMapTask 任务执行的状态信息。通过 MapOutputTracker 的 getStatus 方法获取 map 任务执行的状态信息，得到所需要读取数据的位置。

（2）创建 ShuffleBlockFetcherIterator，经过多次封装，得到 interruptibleIterator，这样可以支持 Shuffle 数据读取任务的中断操作。

（3）基于 interruptibleIterator，根据 dependecy 中的 aggregator 等信息，得到 aggregatedIter（类型为 Iterator[Product2[K, C]]）。

（4）最终判断依赖中是否有定义 keyOrdering，如果没有，则直接返回 aggregatedIter；如果有，则创建 ExternalSorted，在此基础上得到 CompletionIterator 对象。

回到内存管理的主题上，我们来看 Shuffle 的 Write 和 Read 两个阶段对执行内存的使用。

- Shuffle Write：若在 map 端选择普通的排序方式，则会采用 ExternalSorter 进行外排，在内存中存储数据时主要占用堆内执行空间。若在 map 端选择 Tungsten 的排序方式，则采用 ShuffleExternalSorter 直接对以序列化形式存储的数据进行排序，在内存中存储数据时可以占用堆外或堆内执行空间，取决于用户是否开启了堆外内存，以及堆外执行内存是否足够。
- Shuffle Read：在对 reduce 端的数据进行聚合时，要将数据交给 Aggregator 处理，在内存中存储数据时占用堆内执行空间。如果需要进行最终结果排序，则要再次将数据交给 ExternalSorter 处理，占用堆内执行空间。

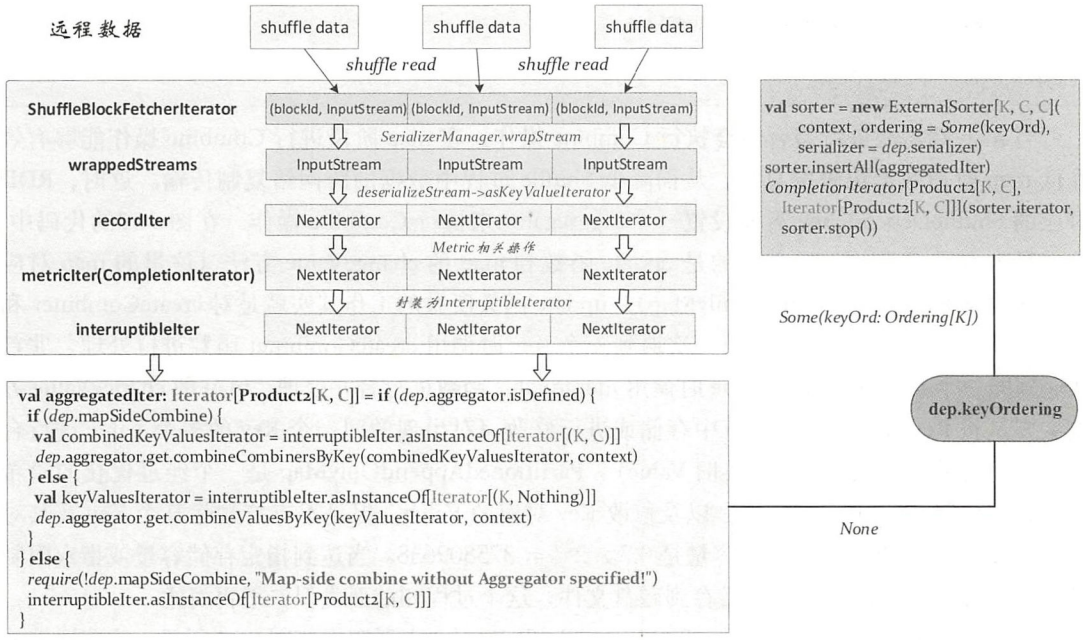


图 9.13　BlockStoreShuffleReader 实现逻辑

在 ExternalSorter 和 Aggregator 中，Spark 会使用一种叫作 AppendOnlyMap 的哈希表在堆内执行内存中存储数据，但在 Shuffle 过程中所有的数据并不能都保存到该哈希表中，这个哈希表占用的内存会进行周期性地采样估算，当其大到一定程度，无法再从 MemoryManager 申请到新的执行内存时，Spark 就会将其全部内容存储到磁盘文件中，这个过程被称为溢存（Spill），溢存到磁盘的文件最后会被归并（Merge）。

9.1.2 Tungsten 内存管理优化基础

催生 Tungsten 内存管理优化的原因主要来自两个方面。

- Java 对象占用内存空间大。相对于 C/C++ 等更加底层的程序语言，Java 对象的存储密度相对偏低。例如，即使最简单的“abcd”字符串，用 Java 的 UTF-16 编码的情况下也需要 8 字节进行存储，加上 Java 内存布局的其他信息（如 header 等），共需要 48 字节的空间来存储“abcd”字符串。
- JVM 垃圾回收（Garbage Collection）的开销大。在海量数据场景下，数据分析通常会涉及转换、清洗、处理等步骤，这个过程伴随着海量的 Java 对象创建与回收，JVM 垃圾回收的执行效率对应用性能有着很大影响。JVM 调优能够在一定程度上提升性能，但是过程

非常烦琐，且需要用户对应用、分布式计算框架和 JVM 都有深入的了解。

Tungsten 优化中堆外内存的引入有效解决了上述问题。9.1.1 小节回顾了 Spark 内存管理方面的基础知识，本节从更细节的实现层面来分析其原理。这部分的实现主要在 common 目录下的 unsafe 包中，如图 9.14 所示，内存分配管理的基础是 MemoryAllocator 接口，定义了 allocate 和 free 两个方法分别来申请内存和释放内存。MemoryAllocator 类的具体实现包括 HeapMemoryAllocator（堆内内存分配）和 UnsafeMemoryAllocator（堆外内存分配）两种。

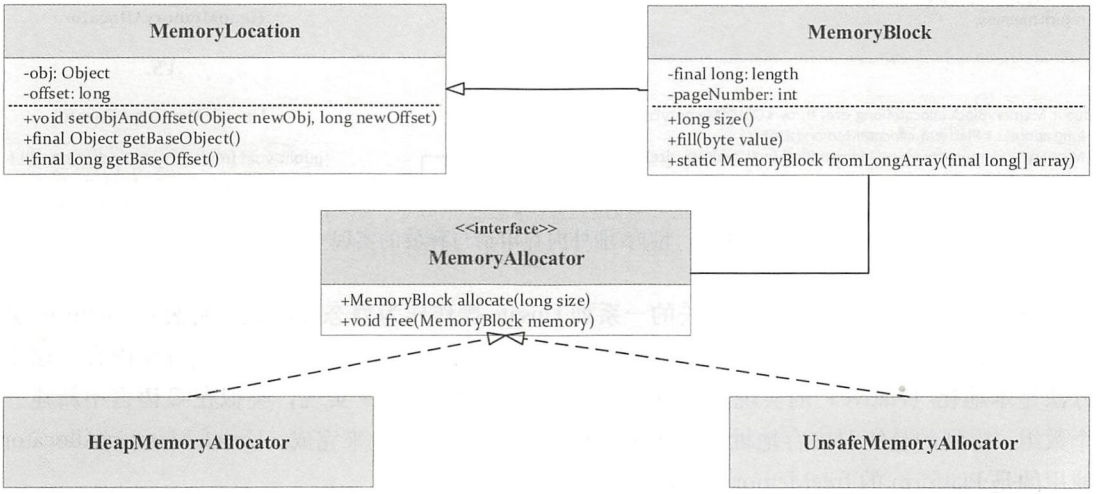


图 **9.14** 堆内/堆外内存管理基础

内存管理必然涉及内存的寻址，因此内存地址的表示方式是重要的部分。堆内内存模式比较常见，所有的 Java 对象都属于这种模式，内存地址由一个 64bit 的对象引用和该对象内部的偏移量（长度）共同表示。另一种则是堆外内存模式，不再由 JVM 托管，而是类似 C/C++ 语言的内存管理，可以采用类似 malloc 方法分配内存。因为摆脱了 JVM，所以可以通过精准的内存管理，有效避免 Java 对象内存占用大和 JVM 垃圾回收开销大的问题。在 Spark 中，堆外内存的访问由 JDK 的 Unsafe 包提供，包括分配内存、释放内存和内存指针操作等。在堆外内存模式下，内存地址可以简单地由一个 64bit 的指针（长整型数字）表示。

从图 9.14 中可以看到，Spark 中定义 MemoryLocation 来统一记录和追踪堆内、堆外的内存地址：在堆内模式中，内存由该对象的引用（obj）和 64bit 的偏移量（offset）去寻址；在堆外模式中，内存可以直接通过 64bit 长度的地址进行寻址（将 obj 设置为 null）。MemoryBlock 用来表示一段连续的内存块，在 MemoryLocation 的基础上增加了内存页编号（pageNumber）和对应数据的大小（length）。

两种 MemoryAllocator 的实现对比如图 9.15 所示。首先是 HeapMemoryAllocator，从 allocate 方法的实现可以看到，以 8 字节对齐的方式申请长度为（size+7）/8 的长整型数组

(array)；然后构造 MemoryBlock 对象，其 obj 成员即为 array，offset 成员为 Platform 类中得到的 LONG_ARRAY_OFFSET 指标。堆内内存的释放一般不做任何操作，直接交给 JVM 的垃圾回收。实际上，当申请的数据量比较大时，会有一个内存池来管理，内存池中预先存放各种尺寸的内存块。

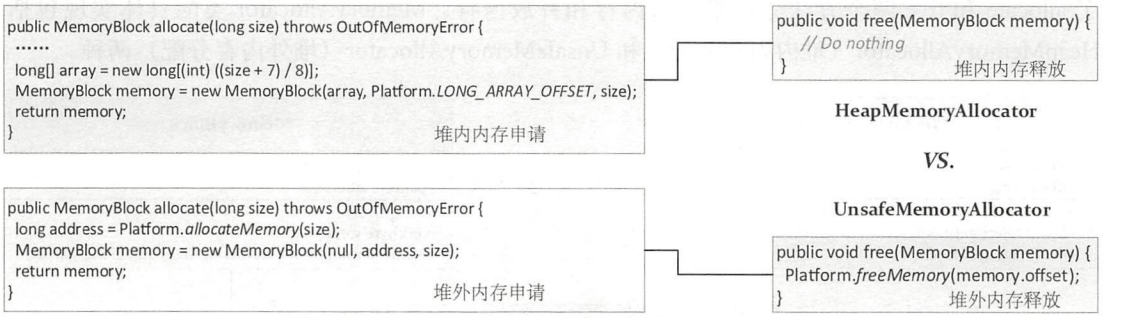


图 9.15　堆内/堆外内存申请与释放的实现

依赖于 Unsafe 包，Spark 将相关的一系列 Unsafe 操作作为静态方法统一封装在 Platform 类中。因此，UnsafeMemoryAllocator 中直接使用 Platform 的 allocateMemory 方法分配内存，这个方法是本地化（Native）的实现，即通过 JNI 最终调用 C 和 C++ 实现，类似在 C 语言中新建一个数组，得到的是绝对内存地址。堆外内存的回收也需要 Spark 来完成，UnsafeMemoryAllocator 使用的是 Platform 的 freeMemory 方法。

在 JDK 的源码中，Unsafe 类会应用多个类，基于其实现可以提高效率。然而，顾名思义，该类是“不安全的”，所分配的内存需要开发人员手动维护。在使用上，Unsafe 这个类的访问是受限的，其构造方法也是私有的。因此，不能通过 new 这种常规的方法来创建其对象，一般都是利用反射机制来获取 Unsafe 实例，Spark 中的实现代码如下。

```
sun.misc.Unsafe unsafe;
    try {
      Field unsafeField = Unsafe.class.getDeclaredField("theUnsafe");
      unsafeField.setAccessible(true);
      unsafe = (sun.misc.Unsafe) unsafeField.get(null);
    } catch (Throwable cause) {
      unsafe = null;
    }
```

实际上，Platform 仅仅是封装 JDK 中的 Unsafe 类，几乎不进行额外的操作。例如，Platform 类中用来分配内存的 allocateMemory 方法就是直接调用 Unsafe 类中的 allocateMemory 方法。值得一提的是，Platform 也通过反射机制来调用 java.nio.Bits 中的 unaligned 方法判断底层系统是否支持非对齐访问（Unaligned-Access）模式。此外，Platform 中也定义了与平台相关的若

干变量，如表 9.2 所示，Unsafe 中的 arrayBaseOffset 方法返回的是数组第一个元素地址相对于数组起始地址的偏移值。除此之外，Unsafe 中还提供了 arrayIndexScale 方法来得到数组一个元素占用的字节数，这样在确定下标 idx 的情况下，可以很方便地获取数组元素的偏移值 (arrayBaseOffset+arrayIndexScalse×idx)。

表 9.2 Platform 相关变量

变量 (偏移量)	调用方法
BOOLEAN_ARRAY_OFFSET	arrayBaseOffset(boolean[].class)
BYTE_ARRAY_OFFSET	arrayBaseOffset(byte[].class)
SHORT_ARRAY_OFFSET	arrayBaseOffset(short[].class)
INT_ARRAY_OFFSET	arrayBaseOffset(int[].class)
LONG_ARRAY_OFFSET	arrayBaseOffset(long[].class)
FLOAT_ARRAY_OFFSET	arrayBaseOffset(float[].class)
DOUBLE_ARRAY_OFFSET	arrayBaseOffset(double[].class)

Tungsten 内存管理机制较核心部分的实现在 TaskMemoryManager 类中，如图 9.16 所示。可以看到，Tungsten 按照内存页表（pageTable）的方式来管理内存，图 9.16 中的 pageTable 本质上是一个 MemoryBlock 的数组，这些内存会被 Task 内部的内存消费者（MemoryConsumer）使用。

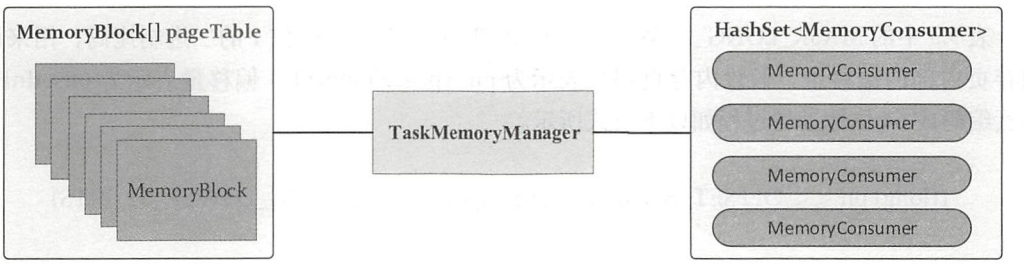


图 9.16 Tungsten 内存管理机制之 TaskMemoryManager

每个 Task 的内存空间被划分为多个内存页（page），每个内存页本质上都是一个内存块 (MemoryBlock)。为统一堆内和堆外的内存访问方式，TaskMemoryManager 引入了类似操作系统中虚拟内存逻辑地址的概念，并将逻辑地址映射到实际的物理地址。逻辑地址由一个 64bits 的长整型表示，其中处于高位的 13bits 用来表示页编号 (page Number)，处于低位的 51bits 用来表示在该内存页内部的偏移 (offsetInPage)。这样内存映射的过程，实际上就是先根据内存页编号查询页表 (pageTable)，得到对应的内存页，然后得到该页的物理地址，最后在物理地址上加上偏移，得到实际内存物理地址。因此，所有内存地址都可由 pageNumber 和 offsetInPage 决定。

Note：对于堆外内存，当给定内存地址时，可以使用与 C/C++ 语言中操作指针一样的方法，在一个数据结构中指向另一个数据结构来访问内存空间中的数据。然而，在 JVM 堆内内存模式

下，GC 会导致堆内结构重新组织，Java 对象的内存地址不是固定不变的，无法直接使用对象指针（地址）来访问。

在介绍内存寻址实现逻辑之前，有必要了解内存管理中的一些变量，如表 9.3 所示。这些变量都是 final 类型，PAGE_NUMBER_BITS 与 OFFSET_BITS 读者已经熟悉，即 64 位中的 13 位与 51 位用来表示内存页的下标与内存页内部偏移量的范围。因此，每个 Task 内部可以最多支持 PAGE_TABLE_SIZE（8192）个内存页，单个内存页最大支持 MAXIMUM_PAGE_SIZE_BYTES 字节的内存空间（注：理论上虽然 OFFSET_BITS 支持大约 2PB 以上的内存，但受限于 JVM 堆，单个内存页上限是 16GB）。所以，Spark 中单个 Task 最多可以支持 $8192 \times 2^{32} \times 8\text{bytes} \approx 35\text{TB}$ 的内存空间寻址。

表 9.3　内存管理重要变量

变量定义	变量说明
PAGE_NUMBER_BITS	内存页索引的二进制位数 (13)
OFFSET_BITS	内存页内部偏移量的二进制位数 (51)
PAGE_TABLE_SIZE	pageTable 的大小 (1 << 13 = 8192)
MAXIMUM_PAGE_SIZE_BYTES	单个 page 最大空间 (((1L << 31) − 1) × 8L)
MASK_LONG_LOWER_51_BITS	51 位 1 的二进制掩码 (0x7FFFFFFFFFFFL)

表 9.3 中的 MASK_LONG_LOWER_51_BITS 变量是包含了 51 位 1 的二进制掩码，用来计算内存页内部的偏移量。假设内存页编号表示为 pn（pageNumber），偏移量表示为 offsetInPage，那么编码成 64 位地址的过程如以下公式所示。

$$((\text{long})\ pn << \text{OFFSET\_BITS}) \mid (\text{offsetInPage} \ \&\ \text{MASK\_LONG\_LOWER\_51\_BITS})$$

这样，如果 pn 为 5，offsetInPage 为 10，从地址 (5, 10) 经过编码后得到的地址就是 5<<51|(10&0x7FFFFFFFFFFFL) = 2621450。

解码的过程和上面正好相反，假设内存地址为 pagePlusOffsetAddress，内存页编号和偏移量分成两步进行，都是位操作，如以下公式所示。需要注意的是，内存页编号 pageNumber 为整型，偏移量为长整型。

$$\text{pageNumber} = (\text{int})\ (\text{pagePlusOffsetAddress} >>> \text{OFFSET\_BITS})$$

$$\text{offsetInPage} = (\text{pagePlusOffsetAddress} \ \&\ \text{MASK\_LONG\_LOWER\_51\_BITS})$$

基于上述分析可知，内存页（page）对应的内存可能来自堆内或堆外。TaskMemoryManager 实现的统一内存编码寻址做到了上层透明，上层逻辑只要操作某段内存地址即可，不需要关心该内存空间来自哪里。

9.1.3 Tungsten 内存优化应用

了解了 Tungsten 堆内和堆外内存的管理方式，还需要重新考虑一些基本的数据类型和实现特定的数据结构，并在此基础上实现 Tungsten 方式的计算执行，包括基本的 Shuffle 过程，以及上层的 Join 和 Aggregation 等。本节首先介绍基本的数据类型、数据结构和一些辅助方法（大部分属于 common 下的 unsafe 包），然后分析若干具体应用。

1. 辅助类：UnsafeAlignedOffset/ByteArrayMethods/BitSetMethods/Murmur3_x86_32

UnsafeAlignedOffset 实际上是针对底层不同硬件环境做的一个操作数据记录（Record）长度的接口封装。众所周知，内存中存储的数据除数据本身外，还会存储数据的大小（长度），这样可以很方便地得到偏移量来访问所需要的数据。一般情况下，表示数据长度的都是 8 字节的长整型以支持对齐访问。然而，在 Spark 可扩充处理器架构的硬件环境下，数据长度采用 4 字节的整型来表示。这种差异性导致了许多情形下内存管理的混乱，例如当数据本身都是 8 字节时，4 字节的偏移量导致对齐特性的丢失。为了 Spark 中的其他模块能够更加方便地利用 unsafe 读写内存数据，UnsafeAlignedOffset 在方法层面统一进行了实现。

```
private static final int UAO_SIZE = Platform.unaligned() ? 4 : 8;
public static void putSize(Object object, long offset, int value) {
  switch (UAO_SIZE) {
      case 4: Platform.putInt(object, offset, value); break;
      case 8: Platform.putLong(object, offset, value); break;
      default: throw new AssertionError("Illegal UAO_SIZE");
    }
}
public static int getSize(Object object, long offset) {
   switch (UAO_SIZE) {
      case 4: return Platform.getInt(object, offset);
      case 8: return (int)Platform.getLong(object, offset);
      default: throw new AssertionError("Illegal UAO_SIZE");
    }
}
```

UnsafeAlignedOffset 中定义了变量 UAO_SIZE，调用 Platform 中的 unaligned 方法，如果为 true，则该值设为 4，否则设为 8。上面代码中的 putSize 方法一般用来向内存中写入数据的长度 (int value)，当 UAO_SIZE 为 4 时，会调用 putInt。此外，UnsafeAlignedOffset 还定义了 getSize 方法，与 putSize 类似，判断不同的情况并调用不同的方法。UnsafeAlignedOffset 类在涉及自定义编码的场景（如 BytesToBytesMap 等）中会大量用到，后续内容中可以见到。

顾名思义，ByteArrayMethods 和 BitSetMethods 中定义的分别是对字节数组 (ByteArray) 和位集合 (BitSet) 的操作方法。ByteArrayMethods 包含 nextPowerOf2、roundNumberOfBytesToNearest-

Word 和 arrayEquals 3 个静态方法。BitSetMethods 则提供了 set、unSet、isSet、anySet 和 nextSetBit 5 个主要功能。相对来讲，这些方法都比较好理解，具体实现可以参见代码。Murmur3_x86_32 基于 Guava 中的 Murmur3_32HashFunction，为 unsafe 范围下的各对象（如 BytesToBytesMap、UnsafeInternalRow 等）提供了计算 hashcode 的功能。Murmur3_x86_32 的实现在此不具体展开。

2. 基本数据类型与数据结构：ByteArray/LongArray/UTF8String

ByteArray 类虽然在 unsafe 包下属于 type 目录，但其本身几乎没有内部定义的属性。ByteArray 作为 final 类型，提供了 writeToMemory、getPrefix 和 subStringSQL 3 个静态的工具方法，其中 writeToMemory 会调用 Platform 类的 copyMemory 方法将字节数组写入特定的内存空间（注：该空间必须已经被分配），getPrefix 从 byte 数组计算得到一个 64 位的整型用于排序。

LongArray 是 Spark 中实现的长整型数组，其构造参数只有 MemoryBlock。相比 JDK 中的数组，LongArray 具有两个方面的特点。

- 支持数据存储在堆内和堆外（与申请到的 MemoryBlock 相关）。
- 没有内置的边界检查，如果关闭 assert 机制，可能会造成 JVM 的崩溃。向 LongArray 写入数据和从 LongArray 读取数据的操作都是调用 Platform 中对应的方法，代码如下（注：代码中省略了边界检查的逻辑）。

```
public void set(int index, long value) {
    Platform.putLong(baseObj, baseOffset + index * WIDTH, value);
}
public long get(int index) {
    return Platform.getLong(baseObj, baseOffset + index * WIDTH);
}
```

这里的 WIDTH 是一个常量，是长整型在内存占用的空间（8 字节），因此如果分配的 MemoryBlock 空间大小为 size，LongArray 的长度就是 size/WIDTH，构造 LongArray 的 MemoryBlock 大小需要满足不超过 Integer.MAX_VALUE×8 的条件。

UTF8String 是 Spark 内部使用的数据类型，负责将 UTF-8 编码的字符串类型编码为字节数组（Array[Byte]），用于字符串的搜索和字符串比较等场景。UFT8String 内部包含 object、offset 和 numBytes 3 个元素，这种表示方式实际上与 MemoryBlock 相似。因此，要构造一个 UTF8String 需要设定这 3 个元素的值。为方便使用，UTF8String 中定义了各种构造方法，包括从字节数组得到 UTF8String 的 fromBytes 方法、从 String 得到 UTF8String 的 fromString 方法等。以 fromBytes 为例，给定 bytes（byte[] 类型，假设不为 null），那么构造得到 UTF8String（bytes, BYTE_ARRAY_OFFSET, bytes.length）对象。

```
private boolean matchAt(final UTF8String s, int pos) {
  if (s.numBytes + pos > numBytes || pos < 0) {
    return false;
  }
  return ByteArrayMethods.arrayEquals(base, offset + pos, s.base, s.offset, s.numBytes);
}
```

类似 JDK 中的 String 对象，UTF8String 提供了各种常用的字符串处理功能，包括基本的 startsWith、endsWith、toUpperCase 等方法。例如，上面的代码就展示了字符串匹配的 matchAt 方法，用来判断当前 UTF8String 从 pos 下标开始的数据是否和另一个 UTF8String 相匹配。这些方法都属于二进制字节上的操作。此外，UTF8String 还实现了 Comparable、Externalizable、KryoSerializable 和 Cloneable 4 种接口，支持大小比较操作、序列化操作和克隆操作。

3. BytesToBytesMap

BytesToBytesMap 可以看作是 Spark 实现的 HashMap，相比 JDK 中的 HashMap，能够有效降低 JVM 对象存储占用的空间和 GC 开销，在 Spark 中占据着重要地位。在 Spark SQL 的各种操作中（包括 Aggregate、Sort 等），对于中间值的存储与处理，几乎都是基于 BytesToBytesMap 数据结构来完成的。BytesToBytesMap 实现机制如图 9.17 所示。

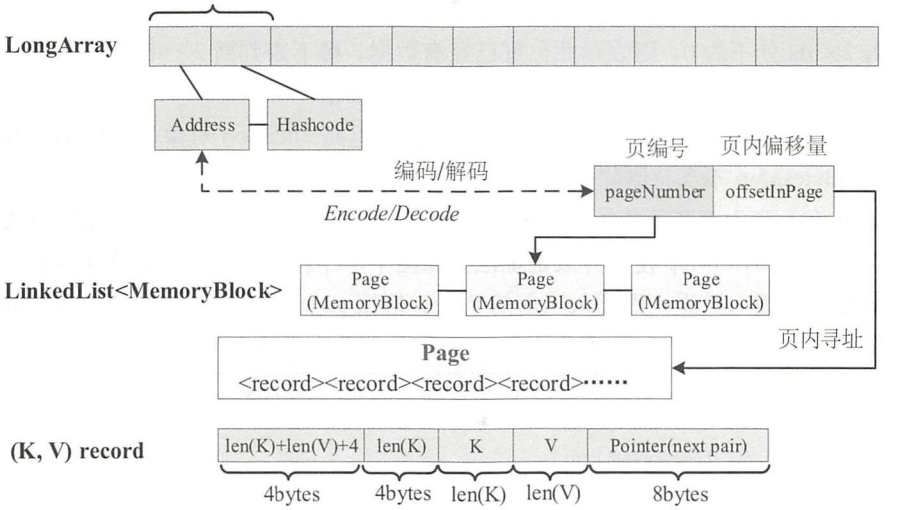


图 9.17 BytesToBytesMap 实现机制

对于每条记录，LongArray 会占用两个 long 分别存放这条记录的 fullKeyAddress 和 hashcode 信息，其中 fullKeyAddress 又包含 pageNum 和 OffsetAddress 两部分。Record 对应的 hashcode，

使用 key 相关信息生成 (keyBase, keyOffset, keyLength)。dataPages 是一个数组，维护申请到的 MemoryBlock。每条记录包含 5 部分信息，以及其占用空间大小，其中 len(k) 和 len(v) 是 8 bytes 的整数倍。最后一部分的指针指向下一个数据，以支持一个 K 对应多个 V 的情况。

```
int uaoSize = UnsafeAlignedOffset.getUaoSize();
final long recordLength = (2 * uaoSize) + klen + vlen + 8;
final Object base = currentPage.getBaseObject();
long offset = currentPage.getBaseOffset() + pageCursor;
final long recordOffset = offset;
UnsafeAlignedOffset.putSize(base, offset, klen + vlen + uaoSize);
UnsafeAlignedOffset.putSize(base, offset + uaoSize, klen);
offset += (2 * uaoSize);
Platform.copyMemory(kbase, koff, base, offset, klen);
offset += klen;
Platform.copyMemory(vbase, voff, base, offset, vlen);
offset += vlen;
Platform.putLong(base, offset, isDefined ? longArray.get(pos * 2) : 0);
```

BytesToBytesMap 数据插入的逻辑和 JDK 中 HashMap 的实现类似，步骤如下。

(1) 根据 Key 相关信息计算 hashcode，利用此 hashcode 得到存储记录 (Record) 的地址下标 idx。

(2) 根据 LongArray 中的存储方式，存放 fullKeyAddress 的是 2×idx。

(3) 获取 fullKeyAddress 的值，若为 0，则表示 2×idx 位置还没有值，返回 Location 对象，Location 包含与下标相关的读取及插入操作。

(4) 若 2×idx 处不为 0，则表示此位置已经有数据，接下来判断 2×idx+1 处存储的 hashcode 和生成的 hashcode 是否相等，以及 key 相关信息是否相等。

(5) 若上一步各种条件判断相等，找到对应位置返回 Location 对象，可以插入及读取数据 (注：BytesToBytesMap 不支持删除数据)。

(6) 若第 (4) 步中判断不相等，表示发生了 hash 冲突，与 JDK HashMap 使用链表解决 hash 冲突不同，BytesToBytesMap 使用开放定址法，探测下一个位置 (idx+1)，重复步骤 (2)。

(7) 若是首次插入数据，则插入当前的 Page，根据 pageNum 和内存地址 OffsetAddress 生成 fullKeyAddress。

```
final long storedKeyAddress = taskMemoryManager.encodePageNumberAndOffset(currentPage,
    recordOffset);
longArray.set(pos * 2, storedKeyAddress);
updateAddressesAndSizes(storedKeyAddress);
longArray.set(pos * 2 + 1, keyHashcode);
```

BytesToBytesMap 支持最多 2^{29} 个 key。2^{30} 是小于 Integer.MAX_VALUE 的 2 的最大指数，由于每条记录 LongArray 需要两个位置存储对应信息，所以最多支持 2^{29}。如果 key 的数量大于 2^{29}，应考虑更加合适的 UnsafeExternalSorter 数据结构。

4. Shuffle 实现：UnsafeShuffleWriter/ShuffleExternalSorter/ShuffleInMemorySorter

Unsafe Shuffle 的实现在一定程度上可以算是 Tungsten 内存管理优化最主要的应用场景。在前面的内容中已经介绍过，Tungsten 方式的 Shuffle 过程中，ShuffleMapTask 的输出数据能够先序列化为二进制数据存储在内存中，再执行相关的操作。Tungsten Shuffle 的写操作由 UnsafeShuffleWriter 完成，与常规的 SortShuffleWriter 的不同之处在于 UnsafeShuffleWriter 中不涉及数据的反序列化操作。

UnsafeShuffleWriter 里面维护着一个 ShuffleExternalSorter，用来进行外部排序，与 SortShuffleWriter 中的 ExternalSorter 功能类似。当 UnsafeShuffleWriter 在逐条写入 RDD 的 (K, V) 记录时，首先会由 Partitioner 根据 K 得到 partitionId，并依次将 K 和 V 序列化写入临时的 serOutputStream 中，然后写入 ShuffleExternalSorter。因此，ShuffleExternalSorter 是 Tungsten Shuffle 写实现的核心所在。

ShuffleExternalSorter 写入数据的过程如图 9.18 所示，可以看到，其内部包括申请到的内存页链表 allocatedPages（LinkedList<MemoryBlock> 类型），每个 Page 中会存储由 (K, V) 数据经过处理后得到的记录（record），currentPage 代表当前的内存页。此外，ShuffleExternalSorter 还需要对数据进行排序，这部分工作由 ShuffleInMemorySorter 即 inMemSorter 完成，其内部包含了一个存储数据指针的 LongArray 数组。

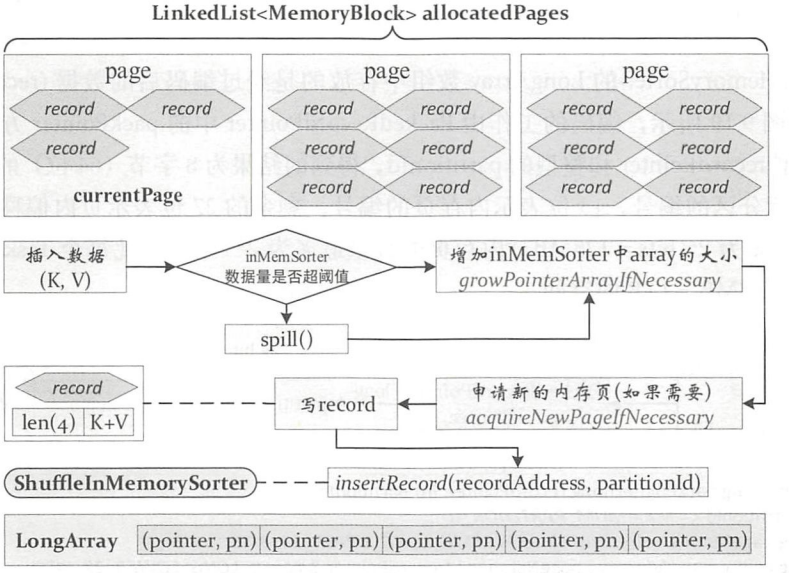


图 9.18 ShuffleExternalSorter 写入数据的过程

当写入一条数据到 ShuffleExternalSorter 中时，首先会检查能否插入到 inMemSorter 中。在

Spark 中，ShuffleInMemorySorter 内存存储的数据有一定的阈值，默认是 1024 × 1024 × 1024 = 1073741824 条，可以通过参数 spark.shuffle.spill.numElementsForceSpillThreshold 设置。因此，如果当前 inMemSorter 中的数据量达到了这个阈值，会首先执行 spill 操作，将内存中的数据写入磁盘，以便释放内存空间。检查了阈值条件之后，还需要调用 growPointerArrayIfNecessary 方法检查 inMemSorter 中的 LongArray 是否有足够的内存空间容纳新数据生成的指针，如果没有，则将 LongArray 的大小扩充到当前大小的两倍。接下来检查当前的内存空间能否存储新的数据（加上存储数据长度的 4 字节），这一步由 acquireNewPageIfNecessary 方法完成，如果无法满足，则 ShuffleExternalSorter 会申请新的内存页。

最终写入数据的实现代码如下。首先调用 Platform 的 putInt 方法向内存空间中插入（K, V）数据的长度，然后调用 Platform 的 copyMemory 方法将（K, V）数据本身写入内存空间中，最终写入的数据是（len+K+V）。此外，在写入数据的同时，会由 TaskMemoryManager 编码得到数据指针 recordAddress，与 partitionId 一起作为参数插入到 ShuffleInMemorySorter 中。至此，ShuffleExternalSorter 写入一条数据的流程完毕。

```
final Object base = currentPage.getBaseObject();
final long recordAddress = taskMemoryManager.encodePageNumberAndOffset(currentPage, pageCursor);
Platform.putInt(base, pageCursor, length);
pageCursor += 4;
Platform.copyMemory(recordBase, recordOffset, base, pageCursor, length);
pageCursor += length;
inMemSorter.insertRecord(recordAddress, partitionId);
```

ShuffleInMemorySorter 的 LongArray 数组中存放的是经过编码后的数据 (recordPointer, partitionId)。如图 9.19 所示，编码的工作由 PackedRecordPointer 中的 packPointer 方法完成，对于输入的长整型 recordPointer 和整型的 partitionId，得到的结果为 8 字节（64 位）的长整型，其中 24 位用来表示分区的编号，13 位表示内存页的编号，剩余的 27 位表示页内偏移量。这样可访问的内存页最多为 2^{27}bits = 128MB，内存页的数量最多为 2^{13} 个，因此每个 Task 中最大的内存可以是 2^{13} × 128MB = 1TB 的空间。

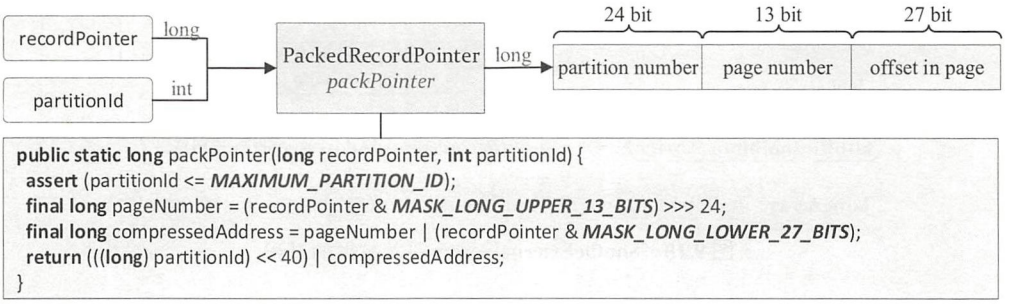


图 9.19　PackedRecordPointer 编码

ShuffleInMemorySorter 完成内存数据排序的方法有两种：一种是基数排序（RadixSort[39]），可以通过参数设置（spark.shuffle.sort.useRadixSort），默认为 true；另一种是 TimSort[40]，对归并算法进行大量的优化。两种排序空间复杂度不一样，因此 ShuffleInMemorySorter 在计算 LongArray 的可用空间（getUsableCapacity）时，会考虑不同的算法。假设 LongArray 长度为 n，如果是 RadixSort，则 LongArray 的可用空间为 $n/1.5$；如果是 TimSort，则可用空间是 $n/2$。

回到 ShuffleExternalSorter，另外一个重要的操作是 spill 文件，该操作由 UnsafeSorterSpillWriter 完成。此时，UnsafeShuffleInMemorySorter 生成一个数据迭代器（getSortedIterator），该迭代器中的数据已经是按照 partitionId 排好序的，迭代器得到的每个元素都是一个指针，对应上面提到的经过 PackedRecordPointer 编码后的地址，通过该地址可以很方便地获取原始数据。需要注意的是，数据在最初写入 UnsafeShuffleExternalSorter 时已经被序列化，因此 spill 操作此时就是单纯地写字节数组。一个文件里不同的 partiton 数据会用 fileSegment 形式来表示，对应的信息存在 SpillInfo 数据结构中，SpillInfo 中包含文件信息、记录每个分区大小的长整型数组和 blockId 3 个元素。此外，spill 操作完成后会释放内存。

回到 UnsafeShuffleWriter，当完成数据的写入之后，会调用 closeAndWriteOutput 方法进行 spill 文件合并操作。文件的合并可以通过 fastMerge 机制来提升效率，将参数 spark.shuffle.unsafe.fastMergeEnabled 设置为 true，如果使用了压缩，需要压缩算法支持 SerializedStreams 的连接，可以直接、简单地将相同分区的压缩数据连接到一起，而且不用进行解压和反序列化操作。

9.2 缓存敏感计算（Cache-aware computation）

缓存敏感计算（Cache-aware computation）是 Tungsten 中一个比较重要的优化，主要对比的是普通的内存计算。在硬件层面，访问 CPU 的 L1/L2/L3 级缓存比访问内存速度快，大数据处理系统可以利用这个特性优化性能。而要将数据存储在 L1/L2/L3 中，数据大小需要满足特定条件。

基于该目标，Tungsten 缓存敏感计算机制通过设计缓存友好的数据结构来提高缓存命中率（Cache hit）和本地化（Cache locality）的特性。到目前为止，Spark 中缓存敏感的计算优化针对的主要是排序（Sort）操作，相应的实现是 UnsafeExternalSorter 和 UnsafeInMemorySorter 类（参见 org.apache.spark.util.unsafe.sort 包）。实际上，cache-aware 排序方式在 Spark 1.5 版本中已经实现[41]，其主要思想受微软 AlphaSort[42] 研究工作的启发。

Tungsten 中 cache-aware 排序原理如图 9.20 所示。常规的做法是每个 record（<key, value>）中有一个指针指向该 record，对两个 record 排序先根据指针定位到实际数据，然后对实际数据进行比较，这个操作涉及的都是内存的随机访问（Random access），缓存本地化（Cache locality）会变得很低。针对该缺陷，缓存友好的存储方式会将 key 和 record 指针放在一起，以 key 为前缀，排序操作是按照线性方式查询 key-pointer 对，避免内存的随机访问。

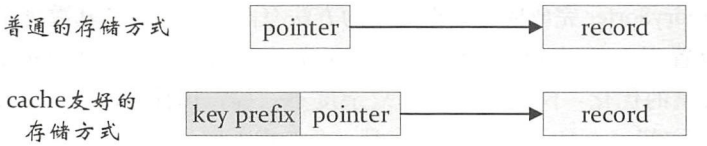


图 9.20　cache-aware 排序原理

在实现上，cache-aware 的排序功能由 UnsafeExternalSorter 和 UnsafeInMemorySorter 来完成。UnsafeExternalSorter 中包含一个 UnsafeInMemorySorter，用于内存中的数据排序，两者之间的关系与 9.1 节中提到的 ShuffleExternalSorter 和 ShuffleInMemorySorter 类似。UnsafeExternalSorter 作为 MemoryConsumer 的子类，可以独立进行内存申请、内存释放和数据 spill 到磁盘的操作。UnsafeExternalSorter 中除各种数据结构跟踪当前的内存使用和分配情况外，还使用链表保存申请到的内存页（allocatedPages），以及使用链表维护 spill 数据到磁盘的对象（UnsafeSorterSpillWriter）。UnsafeInMemorySorter 存储数据指针和数据前缀。对两条记录进行排序时，首先判断两条记录的 prefix 是否相等，如果根据 prefix 就可以判断出两条记录的大小，那么直接返回结果；否则根据数据指针得到的实际数据再进行进一步的比较。相对于实际数据，pointer 和 prefix 占用的空间都比较小，在比较时遍历较小的数据结构更有利于提高 cache 命中率。

创建一个 UnsafeExternalSorter 需要若干参数，如图 9.21 所示，可以分为 3 种。

- Spark 工具类，包括用来完成内存申请与释放的 Task 内存管理器（TaskMemoryManager）、用来存储数据到磁盘的存储管理器（BlockManager）、用来完成序列化数据的序列化管理器（SerializerManager）和 Task 上下文环境（TaskContext）。
- 数据排序支持，包括提供排序前缀比较功能的 PrefixComparator、数据比较功能的 RecordComparator，以及 UnsafeInMemorySorter，其中 UnsafeInMemorySorter 如果为 null，则根据参数自动构造。
- 参数设定，其中 initialSize 用来指定 UnsafeInMemorySorter 的初始大小，pageSizeBytes 是 UnsafeExternalSorter 作为 MemoryConsumer 设定的内存页大小，numElementsForSpillThreshold 表示内存中数据超过该阈值就会触发 spill 操作，canUseRadixSort 表示是否能够使用 RadixSort 来排序。

RecordComparator 是一个抽象类，用来比较两条记录的大小。PrefixComparator 同样是一个抽象类，仅定义了 compare（long prefix1, long prefix2）一个基本的函数。为了支持 Radix 排序，其抽象子类 RadixSortSupport 在此基础上又定义了 sortDescending（是否降序排列）、sortSigned（排序时是否考虑数据的符号位）和 nullsFirst（null 值在顺序上是否放在开头）3 个返回值为 boolean 的方法。在数据排序的过程中，当 PrefixComparator 不适用时，可以使用此比较器对记录进行全维度比较。因为 UnsafeExternalSorter 中涉及 prefix，所以在写入数据前会涉及 prefix 值的计算。给定某个字段的数据，computePrefix 方法可以完成 prefix 计算。

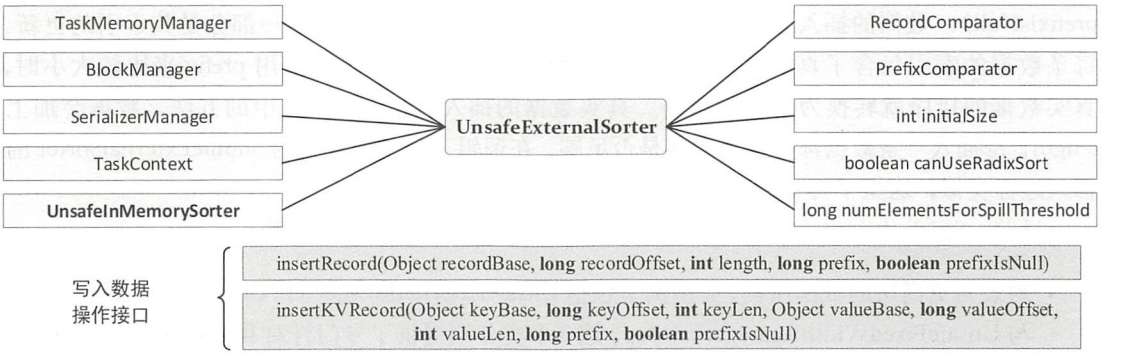


图 9.21 UnsafeExternalSorter 构造与数据写入方法

如表 9.4 所示，BinaryPrefixComparator、StringPrefixComparator 和 DoublePrefixComparator 提供了 computePrefix 功能。

表 9.4 数据前缀计算

提供 computePrefix 类	计算方式
BinaryPrefixComparator	ByteArray.getPrefix
StringPrefixComparator	UTF8String.getPrefix
DoublePrefixComparator	DoublePrefixComparator.computePrefix

所有的数据类型映射可以参见 SortPrefix 类，其 doGenCode 方法中实现的关系如下。

- Boolean 类型，true 为 1L，false 为 0L。
- Integer/Date/Timestamp 类型，直接强制转换为 Long。
- String 类型调用 StringPrefixComparator 中的方法。
- Binary 类型调用 BinaryPrefixComparator 中的方法。
- Decimal 类型精度在一定范围内，强制转换为 Double 类型后调用 DoublePrefixComparator 类中的方法，否则使用 Decimal 的 toUnscaledLong 方法来计算。
- Float/Double 类型，统一为 Double 类型后调用 DoublePrefixComparator 类中的方法。需要注意的是，DoublePrefixComparator 类的 computePrefix 方法实现较为复杂，代码如下。

```
public static long computePrefix(double value) {
      long bits = Double.doubleToLongBits(value);
      long mask = -(bits >>> 63) | 0x8000000000000000L;
      return bits ^ mask;
}
```

总的来看，UnsafeExternalSorter 和 ShuffleExternalSorter 中涉及的各种概念类似，最大的区别在于数据插入方法中是否需要指定前缀信息（包括前缀本身 prefix 和判断前缀是否为 null 的

prefixIsNull)。数据的插入分为两部分，一部分是真实数据的插入，另一部分是其索引的更新。每条数据的索引包含了真实数据插入的地址和其 prefix 值，当记录只用 prefix 来比较大小时，真实数据的排序就转换为索引的排序。真实数据的插入调用 Platform 中的方法，数据会加上 length，每插入一条数据都会检查内存是否足够，在逻辑上和前面分析的 ShuffleExternalSorter 的插入过程类似，这里不再赘述。

目前，Spark 中 UnsafeExternalSorter 数据结构的应用场景有以下几种。

- 封装为更高层的数据结构，典型例子包括 UnsafeExternalRowSorter、UnsafeKVExternalSorter 与 UnsafeFixedWidthAggregationMap，这 3 个类分别提供了专门针对 InternalRow、key-value 数据的排序功能，以及基于 Unsafe 实现的 HashMap 结构，这些数据结构最终都是调用 UnsafeExternalSorter 的方法。
- 计算执行过程中调用，直接使用的场景包括 CartesianProductExec 和 WindowExec 等物理执行计划，用来对数据进行排序。
- 写数据时对数据进行排序，可以参考 SparkHiveDynamicPartitionWriterContainer 和 DynamicPartitionWriteTask 中的使用方式。

9.3 动态代码生成（Code generation）

Spark 引入代码生成主要用于 SQL 和 DataFrames 中的表达式求值（Expression evaluation）[43]。表达式求值的过程是在特定的记录上计算一个表达式的值。当然，这里指的是运行时，而不是在一个缓慢的解释器中逐行做单步调试。对比解释器，代码生成去掉了原始数据类型的封装，更重要的是，避免了昂贵的多态函数调度。

9.3.1 漫谈代码生成

当今绝大多数数据库系统处理 SQL 查询的方式都是将其翻译成一系列的关系代数算子或表达式，然后依赖这些关系代数算子逐条处理输入数据并产生结果。从本质上看，这是一种迭代的模式[44]，某些时候也被称为 Volcano 形式的处理方式，由 Graefe 在 1993 年提出。

该模式可以概括为：每个物理关系代数算子反复不断地调用 next 函数来读入数据元组（Tuple）作为算子的输入，经过表达式处理后输出一个数据元组的流（Tuple stream）。这种模式简单而强大，能够通过任意组合算子来表达复杂的查询。

这种迭代处理模式提出的背景是减轻查询处理的 IO 瓶颈，对 CPU 的消耗则考虑较少[43]。首先，每次处理一个数据元组时，next 函数都会被调用，在数据量非常大时，调用的次数会非常多。最重要的是，next 函数通常实现为虚函数或函数指针，这样每次调用都会引起 CPU 中断

并使得 CPU 分支预测（Branch Prediction）下降，因此相比常规函数的调用代价更大。此外，迭代处理模式通常会导致很差的代码本地化能力，并且需要保存复杂的处理信息。例如，表扫描算子在处理一个压缩的数据表时，在迭代模式下，需要每次产生一个数据元组，因此表扫描算子中需要记录当前数据元组在压缩数据流中的位置，以便根据压缩方式跳转到下一条数据元组的位置。

正是基于这些考虑及实际性能上的观察，一些现代的数据库系统开始尝试摆脱单纯的迭代模式，考虑面向数据块（Block）的方式[45]（一次读取一批数据）来获得数据向量化的处理优势。具体包含两方面，一方面是每次解压一批数据元组，然后每次迭代读取数据时只在这批数据中读取；另一方面是直接在 next 函数读取数据时就读取多个数据元组，甚至一次性读取所有的数据元组。

上述面向数据块的处理方式在一定程度上确实能够消除在大数据量情况下的调用代价，然而在另一方面却丢掉了迭代模式一个重要的优点，即所谓的能够按照管道方式传输数据(Pipelining data)。管道方式意味着数据从一个算子传输到另一个算子不需要进行额外复制或序列化处理。例如，select 就是一个具有管道方式的算子，能够在不修改数据的前提下传输数据元组到下一个算子。然而，在面向数据块的处理方式中，一次产生多条数据元组往往需要序列化保存，虽然能够带来向量化处理的优势[46]，但是破坏了管道方式的优点，并且在通常情况下会消耗更多的内存与网络带宽。

因此，当内存与 IO 不再成为瓶颈后，CPU 成为现代数据库系统的主要瓶颈。MonetDB 系列的数据库处理系统采用的是面向数据块的思路。另一个思路则是将查询编译成中间可执行的格式，即动态代码生成（Code generation），来取代解释性的结构。在实际数据处理中，比较有意思的一个发现是开发人员手写的程序明显比向量化的数据库系统性能高[47]。

总的来讲，动态代码生成能够有效地解决 3 个方面的问题。

- 大量虚函数调用，生成的实际代码不再需要执行表达式系统中统一定义的虚函数（如 Eval、Evaluate 等）。
- 判断数据类型和操作算子等内容的大型分支选择语句。
- 常数传播（Constants propagation）限制，生成的代码中能够确定性地折叠常量。

近年来，在代码生成方面，学术界与企业界也做了大量的工作。例如，文献[48]中将查询逻辑转换为 Java 字节码、HIQUE 系统将查询翻译为 C 代码。代码生成技术在大数据系统中的应用也非常广泛，例如，Impala 中采用 LLVM（Low-Level Virtual Machine）作为中间代码加速数据处理、Spark SQL 生成 Java 中间代码来提升效率。

9.3.2 Janino 编译器实践

在 Spark 中，生成的代码由 Janino 进行编译。Janino 是一个小且快的 Java 编译器，它不仅能像 javac 工具那样将一组源文件编译成字节码文件，还可以对一些 Java 表达式、代码块、类中的文本（Class body）或内存中的源文件进行编译，并把编译后的字节码直接加载到同一个 JVM 中运行。Janino 不是一个开发工具，而是运行时的嵌入式编译器，比如作为表达式求值的翻译器或类似 JSP 的服务端页面引擎。

下面的案例展示了 Janino 中 ExpressionEvaluator 的用法，将符合 Java 语法的字符串表达式（“a+b”）进行编译（Cook），然后加载使用，读者可仔细体会其中的调用方式。

```
import org.codehaus.commons.compiler.CompileException;
import org.codehaus.janino.ExpressionEvaluator;
import java.lang.reflect.InvocationTargetException;

public class ExpressionTest {
    public static void main(String[] args) throws CompileException, InvocationTargetException {
        ExpressionEvaluator ee = new ExpressionEvaluator();
        ee.setParameters(new String[]{"a", "b"}, new Class[]{int.class, int.class});
        ee.setExpressionType(int.class);
        ee.cook("a + b");
        int result = (Integer) ee.evaluate(new Object[]{19, 23});
        System.out.println(result);
    }
}
```

此外，Janino 还支持编译更加复杂的 Java 代码模块。典型的案例如下，method1 和 method2 都是静态方法，ScriptEvaluator 执行 cook 方法来编译整个代码模块，然后调用 evaluate 执行该模块的处理逻辑。

```
import java.lang.reflect.InvocationTargetException;
import org.codehaus.commons.compiler.CompileException;
import org.codehaus.janino.ScriptEvaluator;

public class ScriptTest {
    public static void main(String[] args) throws CompileException,
        NumberFormatException, InvocationTargetException {
        ScriptEvaluator se = new ScriptEvaluator();
        se.cook(
                ""
                        + "static void method1() {\n"
                        + "    System.out.println(1);\n"
                        + "}\n"
                        + "\n"
                        + "method1();\n"
                        + "method2();\n"
```

```
                    + "\n"
                    + "static void method2() {\n"
                    + "    System.out.println(2);\n"
                    + "}\n"
        );
        se.evaluate(new Object[0]);
    }
}
```

这里通过两个简单案例介绍了 Janino 的使用方式，以帮助读者对代码生成的相关工具有一个直观的了解，更加复杂的用法读者可进一步自行探索。

9.3.3 基本（表达式）代码生成

Tungsten 代码生成分为两部分，一部分是最基本的表达式代码生成，另一部分称为全阶段代码生成，用来将多个处理逻辑整合到单个代码模块中。

代码生成的实现中 CodegenContext 可以算是最重要的类，CodegenContext 作为代码生成的上下文，记录了将要生成的代码中的各种元素，包括变量、函数等。如图 9.22 所示，可以将 CodegenContext 中的元素划分为几个大的类别。

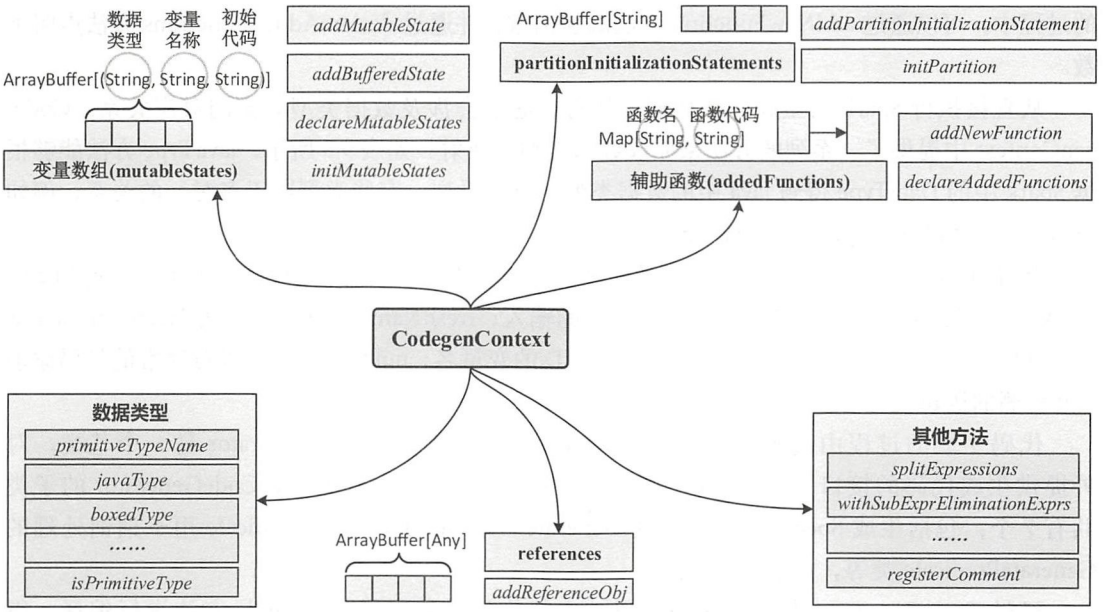


图 9.22 CodegenContext 类

首先是生成的代码中的所有变量 mutableStates，类型为三元字符串 (javaType, variableName, initCode) 构成的数组，其中的字符串分别代表 Java 类型、变量名称和变量初始化代码。例如，

三元组（“int”，“count”，“count = 0;”）将在生成的类中作为成员变量 count 即“private int count;”，同时在类的初始化函数中加入变量初始化的代码即“count = 0”。

变量数组 mutableStates 相关的方法有 4 个，其中 addMutableState 方法用来添加变量，需要指定 Java 类型、变量名称和变量初始化代码。addBufferedState 方法用来添加缓冲变量，与常规的状态变量的不同之处是，缓冲变量一般用来存储来自 InternalRow 中的数据，比如一行数据中的某些列等。因此，这些变量仅在类中声明，但是不会在初始化函数中执行，该方法返回的是 ExprCode 对象。declareMutableStates 方法用来在生成的 Java 类中声明这些变量（默认均为 private 类型）。initMutableStates 方法用来在类的初始化函数中生成变量的初始化代码，输出的元素都是每行一个。

除常见的变量外，对于 Spark RDD 的处理，有些处理逻辑中可能会涉及 RDD 分区的下标（partitionIndex）。针对这些内容，CodegenContext 中也会进行保存。图 9.22 中的 partitionInitializationStatements，作为字符串类型的数组，提供了添加相关代码的方法 addPartitionInitializationStatement 和代码初始化相关的 initPartition 方法。同样的，references 也是一个数组，用来保存生成代码中的对象（objects），可以通过 addReferenceObj 方法添加。

除变量外，生成的代码中另一个比较重要的部分是添加的函数。在 CodegenContext 中，addedFunctions 类型为 Map[String, String]，提供了函数名和函数代码的映射关系。在代码生成的过程中，可以通过 addNewFunction 方法添加函数，并通过 declareAddedFunctions 方法声明函数。

从直接执行 Spark 到生成 Java 代码来执行，必然会涉及数据类型方面的对应关系。CodegenContext 中提供了一系列的方法来完成数据类型的映射。如表 9.5 所示，javaType 方法能够根据 Spark 中的 DataType 得到 Java 中的数据类型。可以看到，有些类型属于多对一的关系，例如 IntegerType 和 DateType 都会生成“int”类型。

此外，CodegenContext 还提供了大量的辅助方法与变量。例如，类型为 Seq[Expr Code] 的 currentVars，用来记录生成的各列作为当前算子的输入；freshName 方法与类型为 HashMap[String, Int] 的 freshNameIds 配合，用来生成具有唯一 ID 的变量名；nullSafeExec 可以对通常的代码添加 null 检测的逻辑。

代码生成的过程由代码生成器（CodeGenerator）完成，CodeGenerator 是一个基类，对外提供生成代码的接口是 generate，如图 9.23 所示。在 Spark SQL 中，CodeGenerator 的子类共有 7 个，包括生成 SpecificOrdering 的 GenerateOrdering 类、生成 Predicate 用于谓词处理的 GeneratePredicate 类等，如表 9.6 所示。

经过 CodeGenerator 类生成后的代码，由其伴生对象提供的 compile 方法进行编译，得到 GeneratedClass 的子类。GeneratedClass 仅仅起到封装生成类的作用，在具体应用时会调用 generate 方法显示地强制转换得到生成的类。

表 9.5 Spark 数据类型与 Java 数据类型

Spark DataType	Java DataType
BooleanType	"boolean"
ByteType	"byte"
ShortType	"short"
IntegerType | DateType	"int"
LongType | TimestampType	"long"
FloatType	"float"
DoubleType	"double"
DecimalType	"Decimal"
BinaryType	"byte[]"
StringType	"UTF8String"
CalendarIntervalType	"CalendarInterval"
StructType	"InternalRow"
ArrayType	"ArrayData"
MapType	"MapData"
udt(UserDefinedType)	javaType(udt.sqlType)
ObjectType(cls) if cls.isArray	javaType(ObjectType(cls.getComponentType))
ObjectType(cls)	cls.getName
其他	"Object"

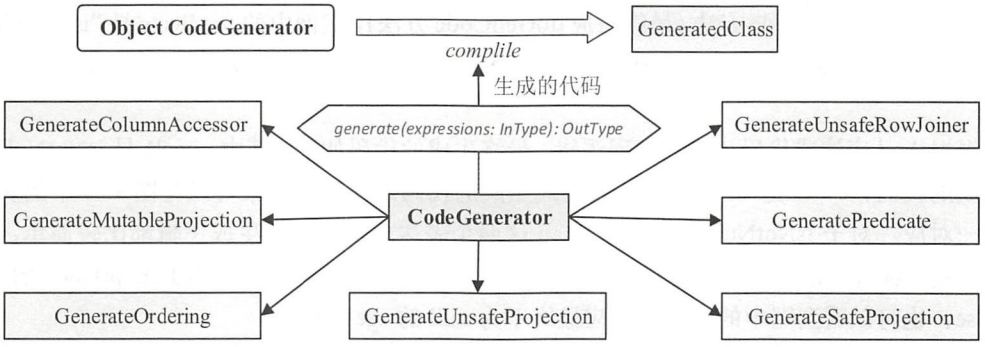


图 9.23 代码生成器 (CodeGenerator)

表 9.6 CodeGenerator 具体实现

CodeGenerator	生成的类
GenerateColumnAccessor	ColumnarIterator
GenerateMutableProjection	MutableProjection
GenerateOrdering	BaseOrdering
GenerateSafeProjection	Projection
GenerateUnsafeProjection	UnsafeProjection
GeneratePredicate	Predicate
GenerateUnsafeRowJoiner	UnsafeRowJoiner

回顾之前的简单案例，如图 9.24 所示，该查询生成的物理计划包括 FileSourceScanExec、FilterExec 和 ProjectExec 3 个节点。为了考察基本的代码生成功能，需要在 Spark 中关闭全阶段代码生成，即将 spark.sql.codegen.wholeStage 设置为 false。

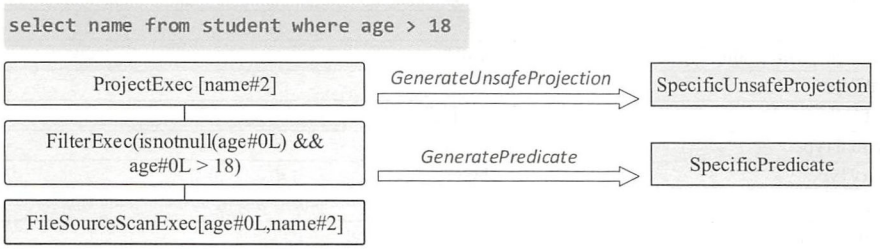


图 **9.24**　代码生成案例

从各个不同节点的 doExecute 方法可知，执行计划中的 FilterExec 节点会调用 GeneratePredicate 对象的 generate 方法生成 Predicate 类，完成过滤算子的逻辑；ProjectExec 节点则会调用 GenerateUnsafeProjection 对象的 generate 方法生成 UnsafeProjection 类，完成投影算子的逻辑。

首先来看 GeneratePredicate 的代码生成过程，如图 9.25 所示（注：图 9.25 中去掉了 generate 函数）。代码生成先构造一个 CodegenContext 对象，然后 FilterExec 算子的谓词表达式 predicate 直接调用其内部的 genCode 方法（最终对应 doGenCode 方法）。案例中的 predicate 是“isnotnull(age) && age > 18”，其根节点为 And 表达式，两个子节点分别为 IsNotNull 表达式和 GreaterThan 表达式。

根据 And 表达式代码生成的实现逻辑，最终生成的代码如下。其中，eval1 是 IsNotNull 表达式生成的代码，eval2 是 GreaterThan 表达式生成的代码。该代码框架可以与图 9.25 中的实际代码一一对应。对于 IsNotNull 和 GreaterThan 这两个表达式，其代码生成逻辑都比较简单，这里不再赘述，读者可以参考具体源代码。因为 IsNotNull 不输出 null 值，所以其 ExprCode 的 isNull 为 false，也可以看到图中的 value4 对应最终不为 null 的 age 字段。

```
${eval1.code}
boolean ${ev.isNull} = false;
boolean ${ev.value} = false;
if (!${eval1.isNull} && !${eval1.value}) {
} else {
  ${eval2.code}
  if (!${eval2.isNull} && !${eval2.value}) {
  } else if (!${eval1.isNull} && !${eval2.isNull}) {
     ${ev.value} = true;
  } else {
     ${ev.isNull} = true;
 }
}
```

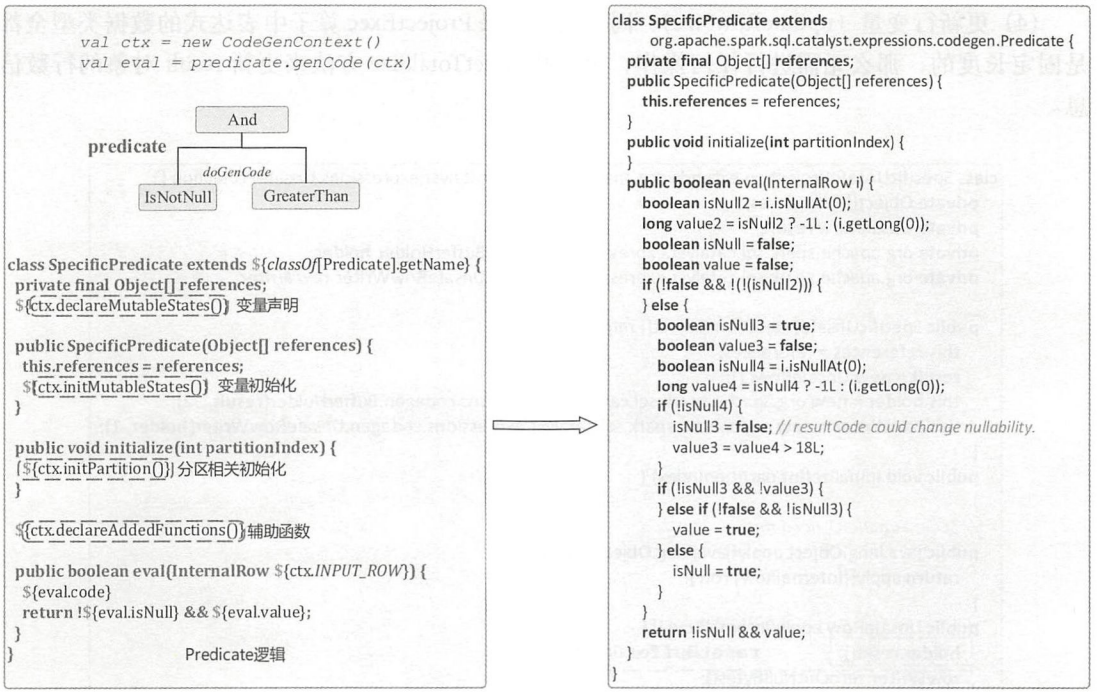


图 9.25 GeneratePredicate 代码生成

接下来，就是 GenerateUnsafeProjection 的代码生成，其代码如图 9.26 所示。实际调用的是 GenerateUnsafeProjection 类中的 create 方法，最终得到 UnsafeProjection 类。GenerateUnsafeProjection 生成的 SpecificUnsafeProjection 类中定义了 3 个变量：首先是 UnsafeRow 类型的 result 变量，用来记录 ProjectExec 的执行结果；其次是 BufferHolder 类型的 holder 变量，即投影处理过程中的缓冲区，用于单数据行的缓存；最后是 UnsafeRowWriter 类型的 rowWriter 变量，用来执行写数据操作。

展开来讲，SpecificUnsafeProjection 类中的 apply 方法完成了以下 4 件事。

（1）重置 BufferHolder 对象（resetBufferHolder）。如果 ProjectExec 算子表达式的数据类型全都是固定长度的，那么此时无需再进行任何操作，否则对 BufflerHolder 执行 reset 操作。

（2）子表达式评估（evalSubexpr）。本例中 CodegenContext 的 subexprFunctions 为空，因此生成的代码不包含任何内容。

（3）写入表达式（writeExpressions）。这部分是 ProjectExec 算子的操作核心，根据数据类型的不同，对应多种情况。图 9.26 展示的是投影算子选择 name 这一列的实际代码。可以看到，首先会判断这一列数据是否为 null，如果是，则直接写入 null；否则获取数据行中对应下标的列，然后用 rowWriter 写入。

(4) 更新行变量（updateRowSize）。同样的，如果 ProjectExec 算子中表达式的数据类型全都是固定长度的，那么无需进行任何操作，否则执行 setTotalSize 方法来更新 result 对象的行数信息。

```
class SpecificUnsafeProjection extends org.apache.spark.sql.catalyst.expressions.UnsafeProjection {
  private Object[] references;
  private UnsafeRow result;
  private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder holder;
  private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter rowWriter;

  public SpecificUnsafeProjection(Object[] references) {
    this.references = references;
    result = new UnsafeRow(1);
    this.holder = new org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder(result, 32);
    this.rowWriter = new org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter(holder, 1);
  }
  public void initialize(int partitionIndex) {
  }
  // Scala.Function1 need this
  public java.lang.Object apply(java.lang.Object row) {
    return apply((InternalRow) row);
  }
  public UnsafeRow apply(InternalRow i) {
    holder.reset();                          resetBufferHolder
    rowWriter.zeroOutNullBytes();
    boolean isNull = i.isNullAt(1);
    UTF8String value = isNull ? null : (i.getUTF8String(1));
    if (isNull) {                                              writeExpressions
      rowWriter.setNullAt(0);
    } else {
      rowWriter.write(0, value);
    }
    result.setTotalSize(holder.totalSize());                   updateRowSize
    return result;
  }
}
```

图 9.26　GenerateUnsafeProjection 代码生成

基本的代码生成过程分析完毕，其他的表达式对应的 Java 代码可以参考源码，并且生成过程类似，都是集中由 7 种 CodeGenerator 在不同情形下触发，然后整合在一起。可以看到，基本的代码生成得到的都是一个个单独的处理模块。

9.3.4 全阶段代码生成（WholeStageCodegen）

Catalyst 全阶段代码生成的入口是 CollapseCodegenStages 规则（参见 QueryExectution）。当设置了支持全阶段代码生成的功能时（默认将 spark.sql.codegen.wholeStage 设置为 true），CollapseCodegenStages 规则会将生成的物理计划中支持代码生成的节点生成的代码整合成一段，

因此称为全阶段代码生成（WholeStageCodegen）。

回顾之前的简单案例，如图 9.27 所示，该查询生成的物理计划包括 FileSourceScanExec、FilterExec 和 ProjectExec 3 个节点。这 3 个节点都支持代码生成，因此 CollapseCodegenStages 规则会在 3 个物理算子节点上添加一个 WholeStageCodegenExec 节点，其主要功能就是将这 3 个节点生成的代码整合在一起。此外，在加入了 WholeStageCodegenExec 物理节点后，物理计划打印输出时不会打印该节点本身，其所囊括的所有子节点在打印输出字符串（generateTreeString）时，都会统一加入特定的（“*”）字符作为前缀，用来区别不支持代码生成的物理计划节点。

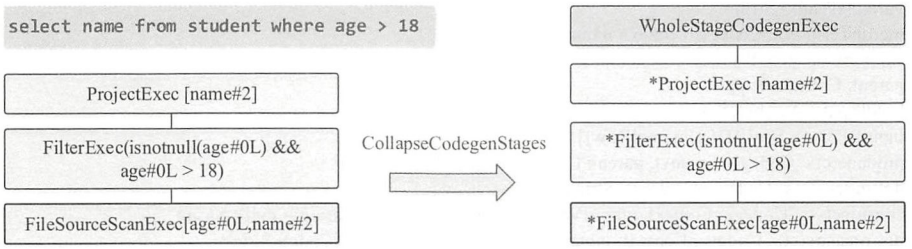


图 9.27 CollapseCodegenStages 规则的作用

一般来讲，对于物理算子树，CollapseCodegenStages 规则会根据节点是否支持代码生成采用不同的处理方式。如图 9.28 所示，在遍历物理算子树时，当碰到不支持代码生成的节点时，会在其上插入一个名为 InputAdapter 的物理节点对其进行封装。

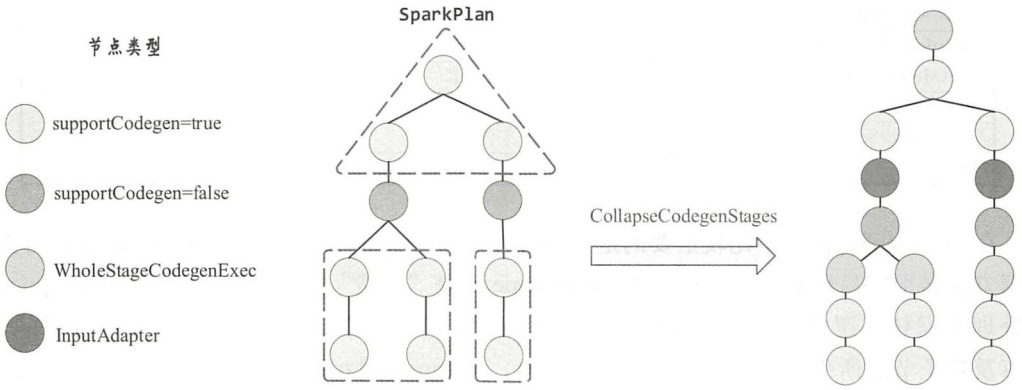


图 9.28 CollapseCodegenStages 规则采用的处理方式

在某种程度上，这些不支持代码生成的节点可以看作是分隔的点，可将整个物理计划拆分成多个代码段。而 InputAdapter 节点可以看作是对应 WholeStageCodegenExec 所包含子树的叶子节点，起到 InternalRow 的数据输入作用。每个 WholeStageCodegenExec 节点负责整合一个代码段，图 9.28 中就插入了 3 个 WholeStageCodegenExec 节点。

在 Spark 中，CodegenSupport 接口代表支持代码生成的物理节点。如图 9.29 所示，CodegenSupport 本身也是 SparkPlan 的子类，提供了 11 个方法和 1 个变量。首先，variablePrefix 返回 String 类型，表示对应的物理算子节点生成的代码中变量名的前缀。不同的节点类型其前缀不同，对应关系如表 9.7 所示。例如，SortMergeJoinExec 节点生成的代码中的变量前缀缩写为“smj”，除特定的缩写外，默认均以 nodeName 的小写作为变量前缀，这样方便彼此之间的区别。

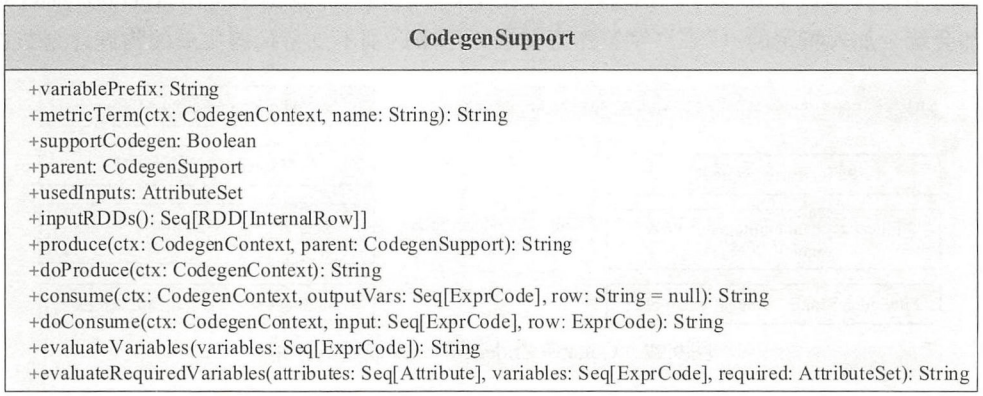


图 9.29　CodegenSupport 接口

表 9.7　SparkPlan 生成代码的变量名前缀

物理计划 SparkPlan	变量名前缀
HashAggregateExec	“agg”
BroadcastHashJoinExec	“bhj”
SortMergeJoinExec	“smj”
RDDScanExec	“rdd”
DataSourceScanExec	“scan”
Other	nodeName.toLowerCase

在 CodegenSupport 中比较重要的是 consume/doConsume 和 produce/doProduce 这两对方法。根据方法名很容易理解，consume 和 doConsume 用来“消费”，返回的是该 CodegenSupport 节点处理数据核心逻辑所对应生成的代码；而 produce/doProduce 则用来“生产”，返回的是该节点及其子节点所生成的代码。在具体实现上，consume 和 produce 都是 final 类型，区别在于 produce 方法会调用 doProduce 方法，而 consume 方法则会调用其父节点的 doConsume 方法。

此外，CodegenSupport 中还保存了其父节点 parent 作为变量，以及其他一些辅助的方法，如判断是否支持代码生成（supportCodegen）、获得产生输入数据的 inputRDDs 等，具体细节这里不再展开。

在大致了解了 CodegenSupport 和 CodegenContext 类的基础上，WholeStageCodegenExec 的执行方式就很好理解了。图 9.30 展示了 WholeStageCodegenExec 的主要逻辑，可以看到其 execute

() 方法具体分为数据获取与代码生成两部分。假设物理算子节点 A 支持代码生成，物理算子节点 B 不支持代码生成，因此 B 会采用 InputAdapter 封装（图 9.30 中的 FakeInput，起到了数据源的作用）。

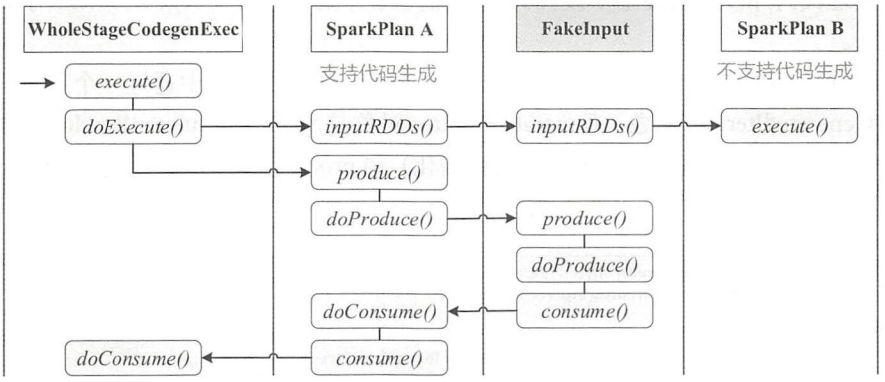


图 9.30 WholeStageCodegen 调用关系

数据的获取比较直接，调用 inputRDDs 递归得到整段代码的输入数据。代码生成可以看作是两个方向相反的递归过程：代码的整体框架由 produce/doProduce 方法负责，父节点调用子节点；代码具体处理逻辑由 consume/doConsume 方法负责，由子节点调用父节点。

节点的 produce 方法调用 doProduce 方法，而 doProduce 中递归调用子节点的 produce 方法，如果是叶子节点或 InputAdapter 节点，doProduce 方法会生成具体的代码框架，因此图 9.30 在 FakeInput 节点的 doProduce 方法中构造代码生成的框架。InputAdapter 中 doProduce 方法的代码如下，可以看到，生成的代码框架基于 while 语句循环，不断地读入数据行（row），并将数据行的处理交给 consume 方法完成。

```
override def doProduce(ctx: CodegenContext): String = {
    val input = ctx.freshName("input")
    ctx.addMutableState("scala.collection.Iterator", input, s"$input = inputs[0];")
    val row = ctx.freshName("row")
    s"""
       | while ($input.hasNext()) {
       |   InternalRow $row = (InternalRow) $input.next();
       |   ${consume(ctx, null, row).trim}
       |   if (shouldStop()) return;
       | }
     """.stripMargin
}
```

每个物理算子节点的 consume 方法将生成相应的代码来完成该节点的数据处理逻辑。consume 方法将递归调用其父节点的 doConsume 方法，这样正好对应了子节点处理逻辑先于父

节点处理逻辑的顺序关系。

WholeStageCodegenExec 生成代码的入口在 doCodeGen 方法中，首先构造一个 CodegenContext 对象；然后将此对象作为 CodegenSupport 中 produce 方法的参数，直接调用 produce 方法生成具体的处理代码片段；最终基于该代码片段和代码生成之后的 CodegenContext 对象，构造完整的代码段。

WholeStageCodegenExec 的代码框架如图 9.31 所示。生成的代码中通过一个 generate 静态方法来构造 GeneratedIterator 对象。GeneratedIterator 对象是 Spark 中 BufferedRowIterator 对象的子类，重载实现了 init 方法（负责相关变量的初始化）和 processNext 方法（用于循环处理 RDD 中的数据行）。

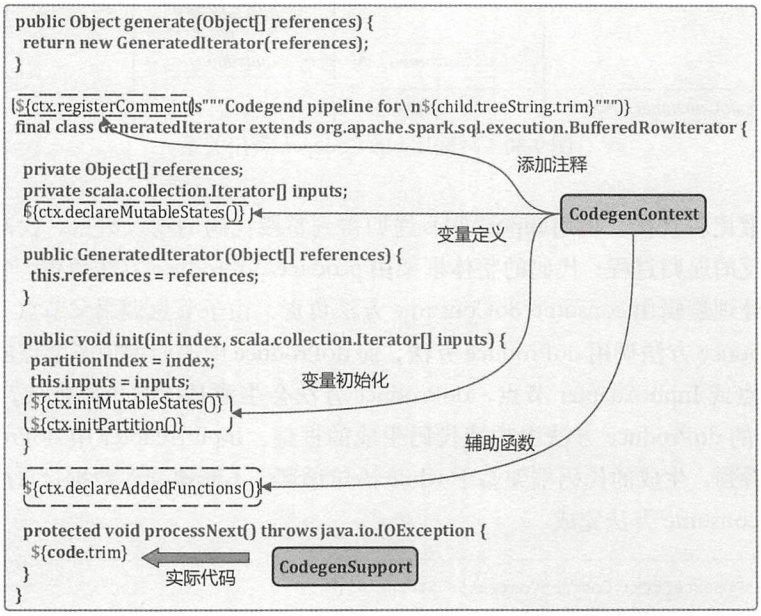


图 9.31　WholeStageCodegenExec 代码框架

可以看到，GeneratedIterator 类中会声明 CodegenContext 中保存的状态变量，在初始化方法 init 中会加入 initMutableStates 与 initPartition 方法。同样的，也会加入 declareAddedFunctions 来声明 CodegenContext 中定义的相关函数。在核心的 processNext 方法中，直接加入 WholeStageCodegenExec 中 produce 方法生成的代码（图 9.31 中的 code.trim）。

后续内容将会详细分析查询实例物理算子树代码生成的具体过程。在此之前，这里先展示由 WholeStageCodegenContext 真正生成的完整 Java 代码，如图 9.32 所示。在每个 WholeStageCodegenContext 节点生成的 Java 代码中，包含一个静态的 generate 方法和一个 GeneratedIterator 类。

```
public Object generate(Object[] references) {
  return new GeneratedIterator(references);
}

final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {

  private Object[] references;
  private scala.collection.Iterator[] inputs;
  private org.apache.spark.sql.execution.metric.SQLMetric scan_numOutputRows;
  private scala.collection.Iterator scan_input;
  private org.apache.spark.sql.execution.metric.SQLMetric filter_numOutputRows;
  private UnsafeRow filter_result;
  private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder filter_holder;
  private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter filter_rowWriter;
  private UnsafeRow project_result;
  private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder project_holder;
  private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter project_rowWriter;

  public GeneratedIterator(Object[] references) {
    this.references = references;
  }

  public void init(int index, scala.collection.Iterator[] inputs) {
    partitionIndex = index;
    this.inputs = inputs;
    this.scan_numOutputRows = (org.apache.spark.sql.execution.metric.SQLMetric) references[0];
    scan_input = inputs[0];
    this.filter_numOutputRows = (org.apache.spark.sql.execution.metric.SQLMetric) references[1];
    filter_result = new UnsafeRow(2);
    this.filter_holder = new org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder(filter_result, 32);
    this.filter_rowWriter = new org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter(filter_holder, 2);
    project_result = new UnsafeRow(1);
    this.project_holder = new org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder(project_result, 32);
    this.project_rowWriter = new org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter(project_holder, 1);
  }

  protected void processNext() throws java.io.IOException {
    while (scan_input.hasNext()) {
      InternalRow scan_row = (InternalRow) scan_input.next();
      scan_numOutputRows.add(1);
      boolean scan_isNull2 = scan_row.isNullAt(0);
      long scan_value2 = scan_isNull2 ? -1L : (scan_row.getLong(0));
      if (!(!(scan_isNull2))) continue;
      boolean filter_isNull2 = false;
      boolean filter_value2 = false;
      filter_value2 = scan_value2 > 18L;
      if (!filter_value2) continue;
      filter_numOutputRows.add(1);
      boolean scan_isNull3 = scan_row.isNullAt(1);
      UTF8String scan_value3 = scan_isNull3 ? null : (scan_row.getUTF8String(1));
      project_holder.reset();
      project_rowWriter.zeroOutNullBytes();
      if (scan_isNull3) {
        project_rowWriter.setNullAt(0);
      } else {
        project_rowWriter.write(0, scan_value3);
      }
      project_result.setTotalSize(project_holder.totalSize());
      append(project_result);
      if (shouldStop()) return;
    }
  }
}
```

图 9.32 查询实例生成代码概览

WholeStageCodegenExec 生成 Java 代码之后，就会交给 Janino 编译器进行编译。主要逻辑如以下代码片段所示：doCodeGen 方法返回 CodegenContext 对象与生成并格式化后的代码（cleanedSource），Spark 首先尝试编译，如果编译失败且配置回退机制（参数 spark.sql.codegen.wholeStage 默认为 true），则代码生成将被舍弃转而执行 Spark 原生的逻辑。编译任务由 CodeGenerator 中的 doCompile 方法执行，调用的是 Jano 中的 ClassBodyEvaluator 对象。需要注意的是，ClassBodyEvaluator 类单独定义了一个 ParentClassLoader，这样避免编译过程中抛出 ClassNotFoundException 异常。最终得到的是一个 GeneratedClass 类，提供了 generate 方法入口供外部调用。

```
val (ctx, cleanedSource) = doCodeGen()
try {
   CodeGenerator.compile(cleanedSource)
} catch {
    case e: Exception if !Utils.isTesting && sqlContext.conf.wholeStageFallback =>
      return child.execute()
}
val references = ctx.references.toArray
val clazz = CodeGenerator.compile(cleanedSource)
val buffer = clazz.generate(references).asInstanceOf[BufferedRowIterator]
val rdds = child.asInstanceOf[CodegenSupport].inputRDDs()
```

如果顺利编译成功，则得到生成的对象（clazz），然后调用其 generate 方法得到 BufferedRowIterator 对象。接下来，WholeStageCodegenExec 的后续处理和其他物理算子节点（mapPartitionsWithIndex）类似。调用 inputRDDs 方法得到 RDD 列表后，会根据 RDD 的数量采取不同的处理逻辑。在现有的实现中，代码生成仅最多支持对两个 RDD 的处理。

上述内容从整体上分析了代码生成的技术实现，为了进一步加深理解，接下来将对实例查询生成代码的详细步骤展开分析。如图 9.33 所示，WholeStageCodegenExec 执行时会调用其子节点 ProjectExec 中的 produce 方法得到生成的代码，因此 ProjectExec 的 produce 方法是整个代码生成过程的入口。

图 9.33 同时标出了完整的调用顺序，ProjectExec 节点的 produce 调用 doProduce 方法，继而调用 FilterExec 节点的 produce 方法。依此类推，一直到叶子节点 FileSourceScanExec 的 doProduce 方法，构造出将要生成的 Java 代码框架。实际上，produce 方法除设置当前 CodegenSupport 节点的 parent 节点和 CodegenContext 的变量前缀外，只是添加注释，然后直接调用 doProduce 方法，代码如下。

```
final def produce(ctx: CodegenContext, parent: CodegenSupport): String = executeQuery {
    this.parent = parent
    ctx.freshNamePrefix = variablePrefix
    s""""
       |${ctx.registerComment(s"PRODUCE: ${this.simpleString}")}
       |${doProduce(ctx)}
```

```
    """.stripMargin
}
```

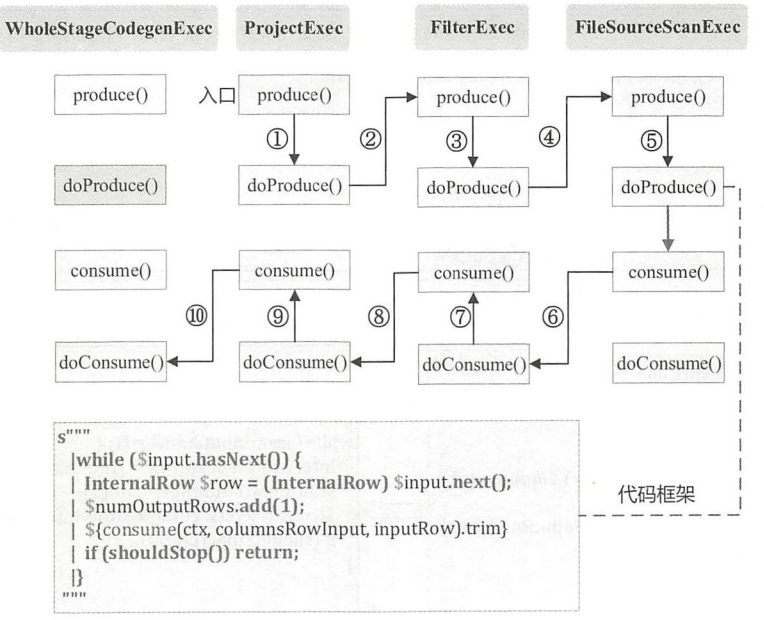


图 9.33 代码生成步骤概览

下面将 WholeStageCodegen 的每一步展开来分析，具体看 FileSourceScanExec 节点的 doProduce 方法所进行的操作，如图 9.34 所示。首先通过 metricTerm 方法在 CodegenContext 中加入一个变量名为 scan_numOutputRows 的 SQLMetric 对象，用来记录从文件中读取的数据行数，以及 Iterator 类型的 scan_input 变量，用来不断读取数据。

此外，当前读取的数据行（InternalRow）在生成的代码中对应命名为 scan_row 的变量，同时将 CodegenContext 对象中的 INPUT_ROW 指向 scan_row 变量。有了该变量，FileSourceScanExec 节点读取的数据列就能够顺利地根据 BoundReference 生成 ExprCode 对象（columnsRowInput 列表）。从图 9.34 中可以看到，除读取数据 scan_row 和更新 scan_numOutputRows 的值外，接下来继续交给 FileSourceScanExec 节点的 consume 方法。

由于是 final 类型，所以所有节点的 consume 处理逻辑均相同。总体来讲，CodegenSupport 对象的 consume 方法所起到的作用是整合当前节点的处理逻辑，构造 (ctx, inputVars, rowVar) 三元组并提交到下一个处理逻辑（父节点的 doConsume 方法）。consume 方法会检查当前生成的代码中是否已经包含了下一步所需的变量，并完成 3 个方面的功能。

- 生成下一步逻辑处理的变量 inputVars，类型为 Seq[ExprCode]，不同的变量代表不同的列。

- 生成 rowVar，类型为 ExprCode，代表整行数据的变量名。
- 在构造上述对象的过程中，相应修改 CodegenContext 对象中的元素。

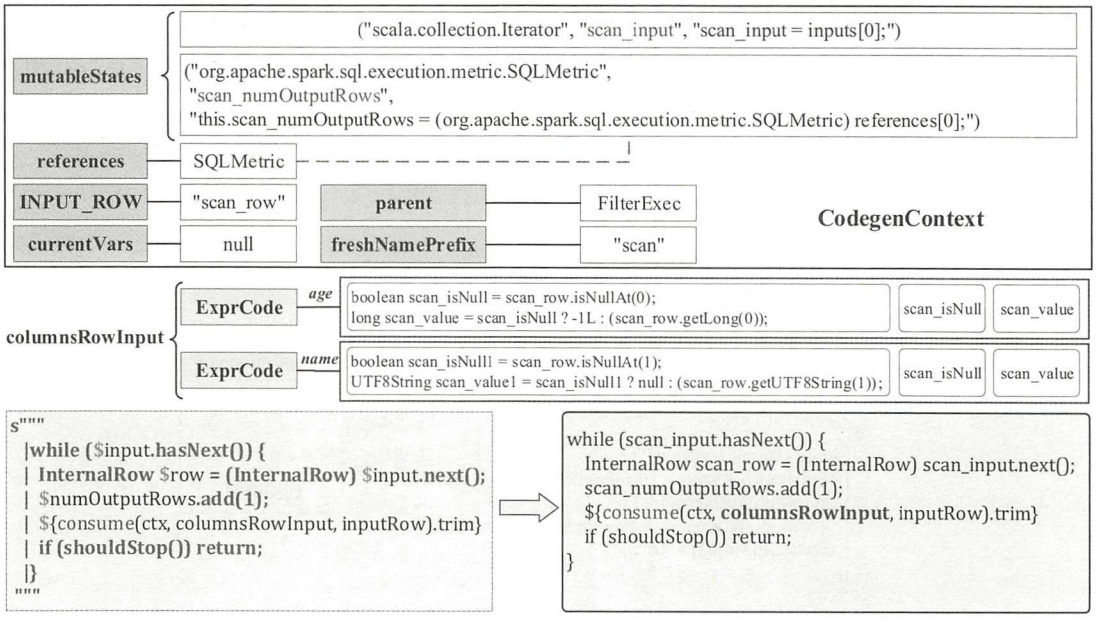


图 9.34　FileSourceScanExec 节点的 doProduce 代码生成

下一步逻辑处理的变量 inputVars 生成逻辑如以下代码所示，其中 row 表示当前数据行对应的变量（ExprCode），outputVars 对应列信息的变量列表（Seq[ExprCode]）。分为两种情况，如果有行变量，那么将 CodegenContext 对象的 INPUT_ROW 指向该行变量，且 currentVars 设为 null，得到的 inputVars 为该节点的输出字段对应的 BoundReference 生成的代码；如果行变量为空，则直接将 outputVars 复制。

```
val inputVars = if (row != null) {
        ctx.currentVars = null
        ctx.INPUT_ROW = row
        output.zipWithIndex.map { case (attr, i) =>
          BoundReference(i, attr.dataType, attr.nullable).genCode(ctx)
        }
      } else {
        outputVars.map(_.copy())
      }
```

类似的，下一步逻辑处理的数据行变量 rowVar 的生成逻辑代码如下。如果传入的行变量不为空，则直接对应该行变量的 ExprCode 对象；如果行变量为空，但是传入的列变量不为空，

那么根据 output 由 GenerateUnsafeProjection 生成代码的主要内容（createCode），否则构造名为 unsafeRow 的 ExprCode 对象。

```
val rowVar = if (row != null) {
      ExprCode("", "false", row)
    } else {
      if (outputVars.nonEmpty) {
        val colExprs = output.zipWithIndex.map { case (attr, i) =>
          BoundReference(i, attr.dataType, attr.nullable)
        }
        val evaluateInputs = evaluateVariables(outputVars)
        ctx.INPUT_ROW = row
        ctx.currentVars = outputVars
        val ev = GenerateUnsafeProjection.createCode(ctx, colExprs, false)
        val code = s"""
          |$evaluateInputs
          |${ev.code.trim}
         """.stripMargin.trim
        ExprCode(code, "false", ev.value)
      } else {
        ExprCode("", "false", "unsafeRow")
      }
    }
```

此外，consume 方法中通常还会用到 evaluateVariables 和 evaluateRequiredVariables 两个辅助函数。这里的 evaluate 直接翻译为“评价”，体现在生成的代码中就是该代码片段已经声明。因此，evaluateVariables 方法得到按行分隔的所有 code 不为空的 ExprCode 代码，并将 ExprCode 对应的 code 设置为空；而 evaluateRequiredVariables 只是根据所需的列集合（AttributeSet）筛选出对应的 ExprCode 代码，其他操作和 evaluateVariables 方法中的逻辑相同。

对应 FileSourceScanExec 算子中的 consume 操作，产生的效果如图 9.35 所示。经过 consume 操作后的变化主要有两个地方：一个是 CodegenContext 对象中的 freshNamePrefix 从“scan”变为“filter”，表示接下来的代码开始进入 FilterExec 的处理逻辑范围；另一个是传递给 doProduce 的输入变量 inputVars，可以看到 scan_value2 和 scan_value3 分别表示 age 与 name 两个字段。

在 consume 的逻辑中，如果传入的变量 row 不为空（对应数据源节点），那么会根据 output 重新得到 BoundReference，这里 FileSourceScanExec 节点就对应这种情况。如果传入的变量 row 为空，那么 inputVars 会对 outputVars 执行 copy 操作，因为这里的 outputVars 变量会用到 CodegenContext 中的 currentVars 来生成 UnsafeRow 代码。

接下来进入到 FilterExec 算子的 doConsume 操作，如图 9.36 所示。同样的，在 FilterExec 中首先会通过 metricTerm 方法在 CodegenContext 中加入一个变量名为 filter_numOutputRows 的 SQLMetric 对象，用来记录经过过滤处理之后的数据行数。此外，CodegenContext 中的 currentVars 会设置为当前传递进来的输入变量（scan_value2 和 scan_value3）所对应的 ExprCode 对象，其

parent 对象在 produce 方法中也会对应变为 ProjectExec 算子。

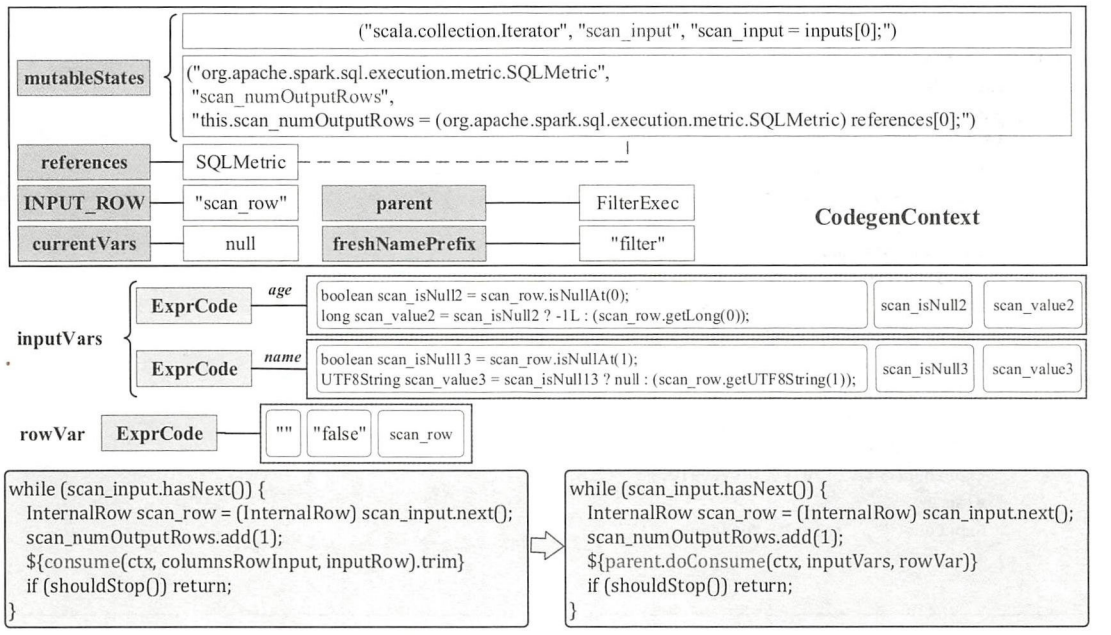


图 9.35　FileSourceScanExec 节点的 consume 代码生成

从图 9.36 中可以看到，FilterExec 算子的 doConsume 方法实际上完成了 4 件事，分别是实际过滤条件的代码生成（Predicated）、null 检测的代码生成（nullChecks）、SQLMetric 变量更新（numOutput.add(1)）和 consume 方法的调用。SQLMetric 变量更新与 consume 方法的调用都比较简单，生成的代码很直观，如图 9.36 所示。

过滤条件与 null 检测是 FilterExec 算子代码生成的核心。在 FilterExec 算子中，会将过滤谓词分为 notNullPreds 和 otherPreds 两部分，notNullPreds 是所有的 IsNotNull 表达式，otherPreds 对应其它的过滤条件。例如，本例中的 notNullPreds 列表中只有一个表达式 isnotnull(age)，otherPreds 列表中也只有一个过滤条件“age > 18”。

具体来看，FilterExec 算子 doConsume 方法中实际过滤条件的代码生成（Predicated）针对的是 otherPreds 谓词，即“age > 18”过滤条件。需要注意的是，这里也会对过滤条件中的字段（age）进行 null 检查，并将记录某个字段是否执行了 null 检查的布尔数组（generatedIsNotNullChecks）的下标设置为 true，避免 null 检测的代码生成（nullChecks）再次执行。图 9.36 中对 age 字段的 null 检查代码已经在 predicated 中体现，因此 nullChecks 为空。

深入 predicated 的具体实现，前 3 行（一直到第一个 continue）负责执行 null 的检查，剩下的 5 行代码负责表达式（age > 18）的逻辑。此外，传递给 consume 方法的 resultVars 中，age 对

应的 scan_value2 已经属于非 null 的字段，因此将其 isNull 设置为“false”；并且 scan_value2 相关的变量在代码中已经声明了（evaluateRequiredVariables），后面不需要再次出现，因此将其 code 设置为空。

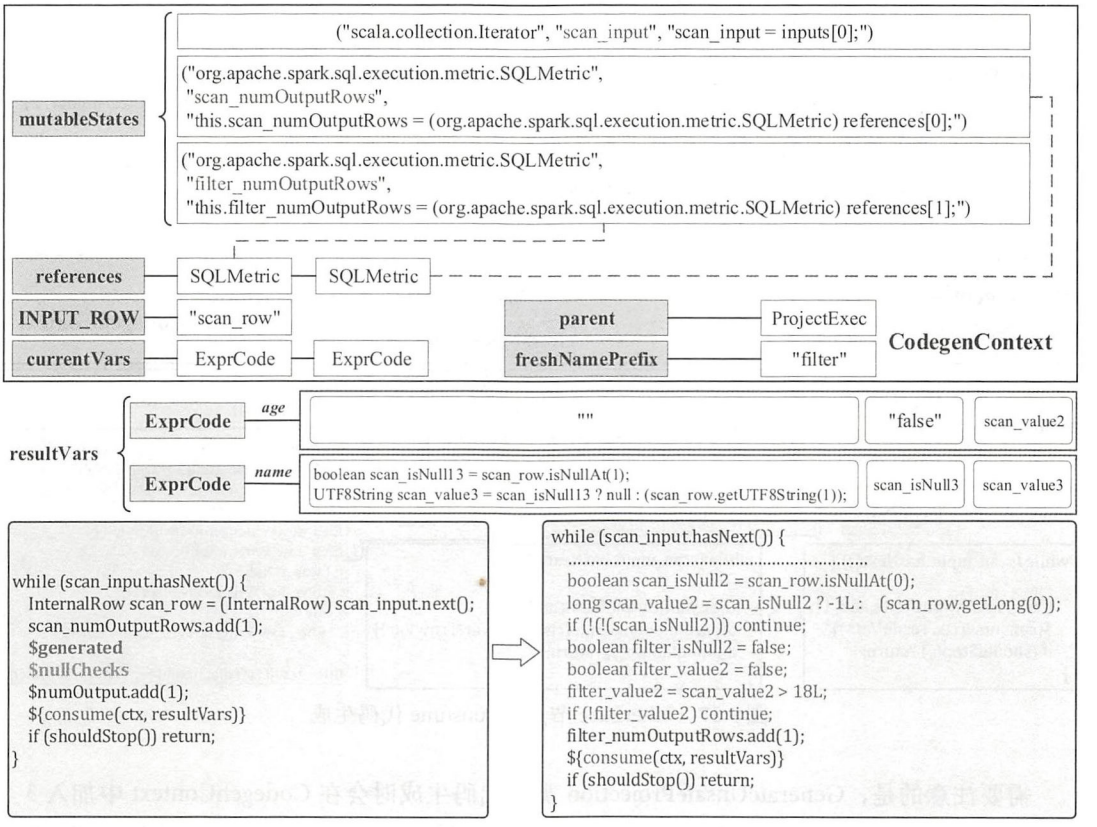


图 9.36 FilterExec 节点的 doConsume 代码生成

FilterExec 算子完成了 doConsume 操作之后进入 consume 方法的操作，如图 9.37 所示。前面已经提到，consume 方法中主要完成 3 件事。首先传递给父节点 doConsume 方法的 inputVars 变量，对于 FilterExec 节点来讲，传递的 row 为空，因此 inputVars 变量是 outputVars 的复制，从生成的代码来看两者完全相同。

FilterExec 算子中的 consume 方法由 GenerateUnsafeProjection 类得到输出的行（rowVar）并传递给 ProjectExec 节点。跟踪其实现，可以知道 CodegenContext 中的 INPUT_ROW 会被设置为当前传入的 row 即 null，而 currentVars 则会被设置为 outputVars 变量。从图 9.37 右下角来看，rowVar 生成的代码包括两部分：输入列（age 与 name 列）变量的声明（evaluate），因为 age 列在之前已经声明过了，code 为空，因此这里会加上 name 列的相关代码；GenerateUnsafeProjection

类生成的代码，这部分内容在前面已经分析过。

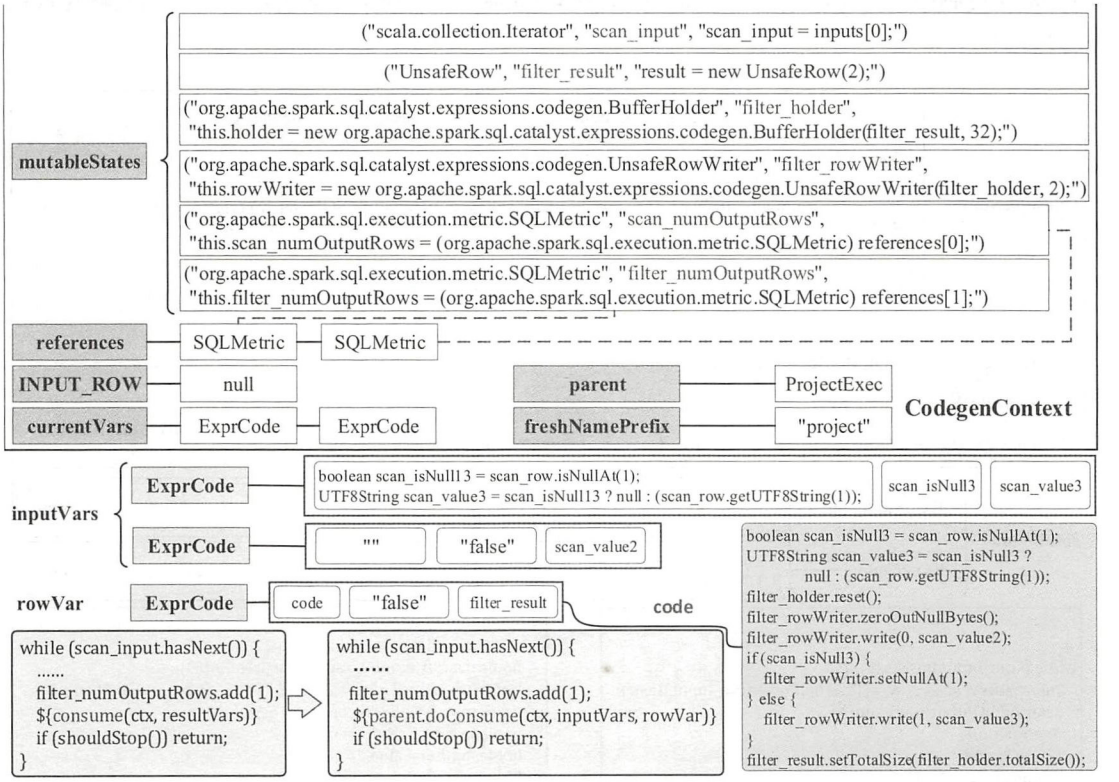


图 9.37　FilterExec 节点的 consume 代码生成

需要注意的是，GenerateUnsafeProjection 类在代码生成时会在 CodegenContext 中加入 3 个变量（filter_holder、filter_rowWriter 和 filter_result）。经过这一步后，最终可以看到，FilterExec 算子的 consume 方法中得到的 evaluated 为空，因此在最终的代码中没有加入多的内容。

接下来进入 ProjectExec 算子的 doConsume 方法的操作中，该方法的参数 input 为 age 和 name 两个变量对应的 ExprCode（scan_value2 和 scan_value3），参数 row 对应上一步 FilterExec 得到的 filter_result 变量。实际上，ProjectExec 的 doConsume 方法用不到 row 参数。

如图 9.38 所示，这一步所做的操作也非常简单，根据需要输出的列（projectList）生成对应的代码，并封装为 ExprCode（resultVars）。这里的 resultVars 只包含 name 列，对应的变量为 scan_value3。最终输出的代码照样会执行 evaluateRequiredVariables 方法（针对的是不确定性的表达式），确保所有的变量都已经声明过了。因为这里不确定性表达式为空，所以不会输入额外的代码。因此，ProjectExec 节点的 doConsume 方法基本上没做比较复杂的事情，只是生成 resultVars，作为参数传递给 consume 方法。

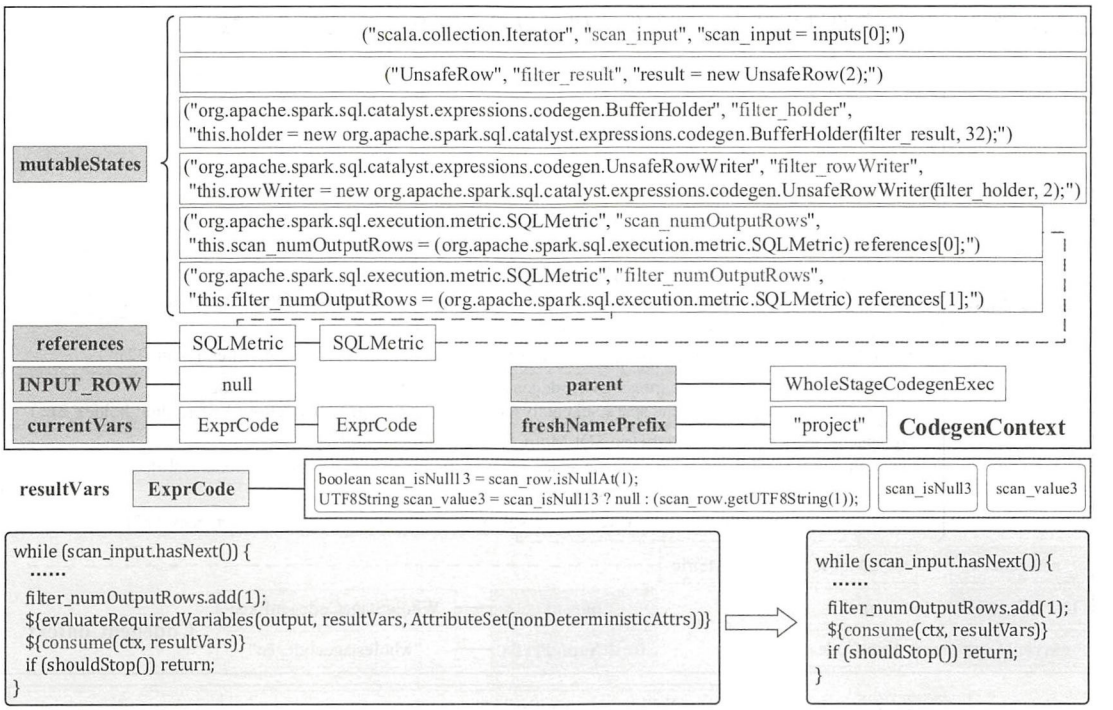


图 9.38 ProjectExec 节点的 doConsume 代码生成

ProjectExec 算子的 consume 方法的代码生成如图 9.39 所示。类似 FilterExec 算子的效果，inputVars 直接复制传入的 outputVars 参数（doConsume 中得到的 resultVars 变量）。同样的，consume 方法中生成的行变量（rowVar）由 GenerateUnsafeProjection 类生成。在此过程中，同样会在 CodegenContext 中增加 project_result、project_holder 和 project_rowWriter 3 个变量，同时将 currentVars 设置为 outputVars，并将变量字符串前缀改为父节点的名称（wholestagecodegen）。可以看到，rowVar 对应的代码和 FilterExec 节点执行 consume 方法时的代码类似。最终的结果是，ProjectExec 节点不直接产生代码，而是将得到的 inputVar 和 rowVar 传递给父节点的 doConsume 方法。此外，也可以看到，ProjectExec 算子不会在 CodegenContext 中构造 SQLMetric 变量记录输出的数据行数，这一点与 FileSourceScanExec 节点和 FilterExec 节点不同。实际上，在一些特殊的算子（例如 Aggregation 与 Join）中，都会添加辅助变量来记录一些关键信息。

全阶段代码生成（WholeStageCodegen）的最后一步都会落脚在 WholeStageCodegenExec 算子的 doConsume 方法。如图 9.40 所示，生成的代码首先会输出 row 变量的 code，对应 ProjectExec 中得到的 project_result 变量。

需要注意的是，是否对得到的结果执行 copy 操作取决于 CodegenContext 对象中的 copyResult 变量。在上述例子中，不需要 copy 操作，因此最终添加结果的代码为 append (project_result)。

至此，整个代码生成的过程结束，完整的代码可以参看图 9.32 中的各个函数。

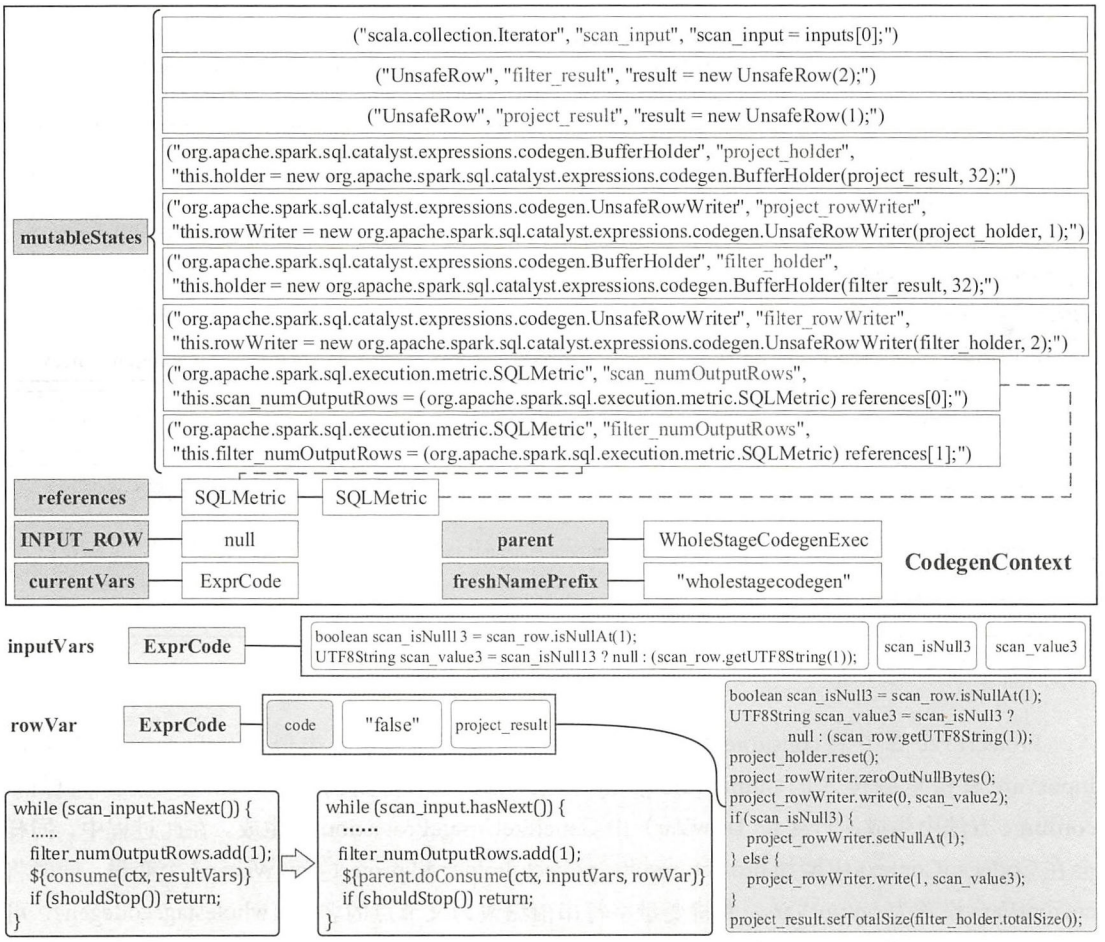


图 9.39　ProjectExec 节点的 consume 代码生成

代码生成是数据处理中一项重要的技术，本节对 Spark SQL 中的具体实现进行了详细、逐步的分析。一般情况下，代码生成技术可以加速（接近一个量级）多种 TPC-DS 查询。目前社区的开发方向是让代码生成可以应用到所有的内置表达式上。此外，未来工作将提升代码生成的等级，从每次一条记录表达式求值到向量化表达式求值，使用 JIT 开发更好的作用于新型 CPU 的指令流水线，从而同时处理多条记录。除通过表达式求值优化内部组件的 CPU 效率外，还会将代码生成推广到更广泛的地方，其中一个就是 Shuffle 过程中将数据从内存二进制格式转换到 wire-protocol。如前文所述，Shuffle 通常会因数据序列化出现瓶颈。通过代码生成，可以显著地提升序列化吞吐量，从而反过来作用到 Shuffle 网络吞吐量的提升。

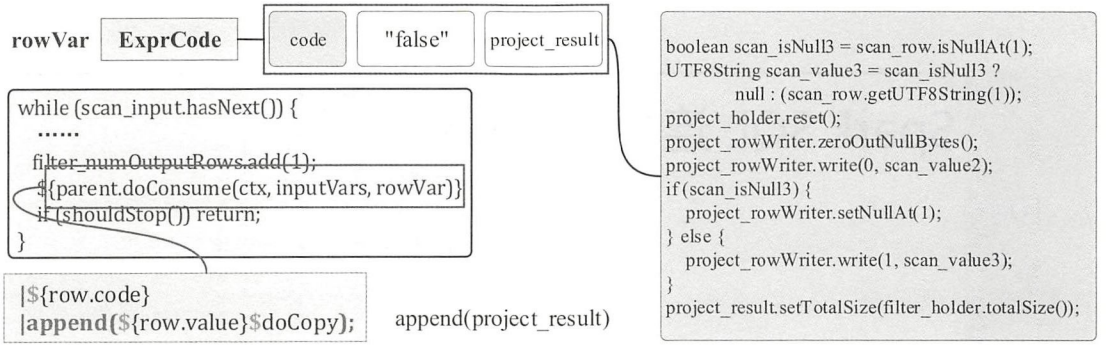


图 9.40 WholeStageCodegenExec 节点的 doConsume 代码生成

9.4 本章小结

本章介绍了 Tungsten 技术的实现原理、包括高效内存管理、缓存友好的排序优化，以及代码生成机制。内存管理方面又具体分为存储内存和执行内存，代码生成包括表达式代码生成和全阶段代码生成。相比前面的章节，本章内容更多的是偏向底层的细节，因此相对琐碎。作为 Spark SQL 的技术核心，其实现方式是社区一直关注的重点，会随着版本的发展而不断优化。

第10章 Spark SQL 连接 Hive

从时间上看，Hadoop 是最早流行的开源大数据处理系统，应用最为广泛。而作为 Hadoop 之上的数据仓库解决方案 Hive，如今已经发展多年，是各企业大数据平台中广泛采用的方案。随着各种新兴的大数据处理技术的蓬勃发展，如何兼容 Hive 中的数据与业务，顺利地迁移到新的计算框架，逐渐摆脱 MapReduce 的限制，都是企业内部技术演化过程中需要考虑的。

目前业界主流的 SQL 引擎都支持与 Hive 的连接，Spark 也不例外。在 Spark 2.0 版本之前，Spark SQL 与 Hive 的连接中除 Hive 元数据（Metastore）外，还依赖 Hive 完成语法解析。从 Spark 2.0 版本开始，Spark SQL 引入 ANTLR 4 作为 SQL 编译器，摆脱了对 Hive 编译器的依赖。本章将重点介绍这部分的实现原理。

10.1 Spark SQL 连接 Hive 概述

从 Spark 2.0 版本开始，连接 Hive 不再需要创建单独的 HiveContex，而是在 SparkSession 中通过 enableHiveSupport 方法开启 Hive 的支持。下面是 example 目录中连接 Hive 的简单应用代码。可见在 Spark SQL 连接 Hive 场景下，不再需要和 DataFrame 等 API 打交道，所有的处理与查询都直接通过 SQL 语句完成。

```
val spark = SparkSession.builder().master("local").enableHiveSupport().getOrCreate()
spark.sql(''CREATE TABLE IF NOT EXISTS src (key INT, value STRING)'')
spark.sql(''LOAD DATA LOCAL INPATH 'kv1.txt' INTO TABLE src'')
spark.sql(''SELECT * FROM src'').show()
```

在这个例子中，当 SparkSession 中开启了 Hive 支持后，配置信息 conf 中会将 Catalog 信息（spark.sql.catalogImplementation）设置为“hive”，这样在 SparkSession 根据配置信息反射获取 SessionState 对象时就会得到与 Hive 相关的对象。第 3 章已经介绍过，Spark SQL 本身的核心类是 SessionState，该类中整合了 Catalog、Parser、Analyzer、SparkPlanner 等所有必需的对象。同样的，在连接 Hive 时，也存在类似的对象 HiveSessionSate，如图 10.1 所示。

为了支持 Spark SQL 连接 Hive，在 HiveSessionState 中主要重载了 3 个对象：Catalog、Analyzer 和 SparkPlanner。其中，Catalog 具体实现为 HiveSessionCatalog，Analyzer 中加入了 Hive 相关的分析规则，SparkPlanner 中加入了 Hive 相关的策略，其他的部分（如 Parser 等）则直接复用 Spark SQL 本身的对象。

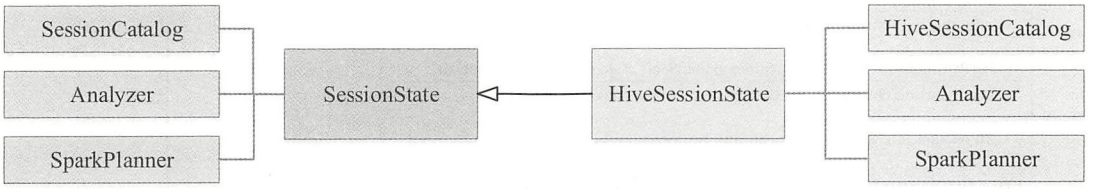


图 10.1 Spark SQL 连接 Hive 概述

10.2 Hive 相关的规则和策略

从实现层面看，Spark SQL 连接 Hive 的主要工作体现在元数据、分析规则和转换策略方面。本节将从这 3 个方面详细介绍 HiveSessionCatalog 体系、Hive-Specific 分析规则和 Hive-Specific 转换策略。

10.2.1 HiveSessionCatalog 体系

在逻辑计划阶段已经总结过 Spark SQL 中默认的 Catalog 体系，对于其他模块，SessionCatalog 提供调用的接口，而 ExternalCatalog 则是实际存储与操作数据的根本所在。在 Hive 场景下，HiveSessionCatalog 继承了 SessionCatalog，HiveExternalCatalog 则继承了 ExternalCatalog，如图 10.2 所示。

在 HiveExternalCatalog 中，对数据库、数据表、数据分区和注册函数等信息的读取与操作都通过 HiveClient 完成。顾名思义，HiveClient 是用来与 Hive 进行交互的客户端，在 Spark SQL 中是定义了各种基本操作的接口，具体实现为 HiveClientImpl 对象。然而在实际场景中，因为历史遗留的原因，往往会涉及多种 Hive 版本，为了有效地支持不同版本，Spark SQL 对 HiveClient 的实现由 HiveShim 通过适配 Hive 版本号（HiveVersion）来完成。

Hive 版本与对应的 Shim 实现关系如表 10.1 所示。需要注意的是，Hive 版本均默认对应最后一个版本号（如 hive.v13 默认对应“0.13.1”版本号）。HiveClient 的 Shim 实现中 Shim_v0_12 继承自 Shim，Shim_v0_13 继承自 Shim_v0_12，依此类推。在 Spark 2.1 中，默认的 Hive 版本为“1.2.1”，因此 Shim 的具体实现为 Shim_v1_2 版本。

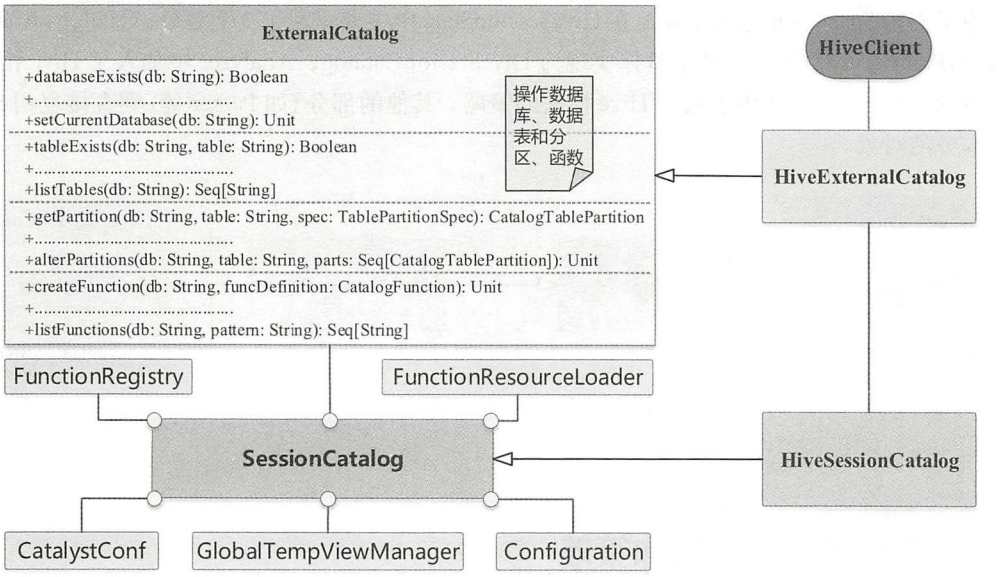


图 10.2　HiveSessionCatalog 体系

表 10.1　HiveClient 不同版本的实现

版本号	Hive 版本	HiveClient Shim 实现
“12”	“0.12”	“0.12.0”	hive.v12	Shim_v0_12
“13”	“0.13”	“0.13.0”	“0.13.1”	hive.v13	Shim_v0_13
“14”	“0.14”	“0.14.0”	hive.v14	Shim_v0_14
“1.0”	“1.0.0”	hive.v1_0	Shim_v1_0
“1.1”	“1.1.0”	hive.v1_1	Shim_v1_1
“1.2”	“1.2.0”	“1.2.1”	hive.v1_2	Shim_v1_2

真正创建 HiveClient 的操作位于 IsolatedClientLoader 类中。一般情况下，Spark SQL 只会通过 HiveClient 访问 Hive 中的类，为了更好地隔离，IsolatedClientLoader 将不同的类分成 3 种，不同种类的加载和访问规则各不相同。

- 共享类（Shared classes）：包括基本的 Java、Scala、Logging 和 Spark 中的类，这些类通过当前上下文的 ClassLoader 加载，调用 HiveClient 返回的结果对于外部来说是可见的。
- Hive 类（Hive classes）：通过加载 Hive 的相关 Jar 包得到的类。默认情况下，加载这些类的 ClassLoader 和加载共享类的 ClassLoader 并不相同，因此，无法在外部访问这些类。
- 桥梁类（Barrier classes）：一般包括 HiveClientImpl 与 Shim 类，在共享类与 Hive 类之间起到了桥梁的作用，Spark SQL 能够通过这个类访问 Hive 中的类。每个新的 HiveClientImpl 实例都对应一个特定的 Hive 版本，如图 10.3 所示。

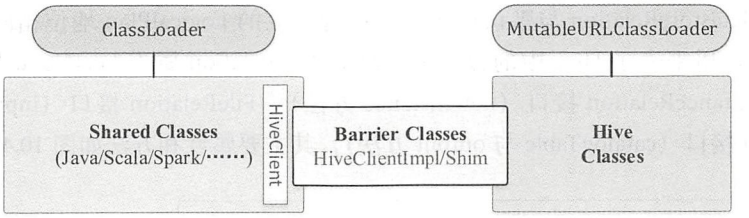


图 10.3 HiveClient 访问的隔离

实际上，IsolatedClientLoader 类能够通过 isolationOn 这个 Boolean 类型值控制是否单独创建 ClassLoader 来加载 Barrier 类。默认情况下为 true，HiveClientImpl 通过创建的 MutableURLClassLoader 加载，以此达到隔离的目的；否则，直接采用当前上下文环境的 ClassLoader 加载。IsolatedClientLoader 创建的 ClassLoader 加载不同类的逻辑如下面的算法所示，不同的情况加载方式不同。

Algorithm 6 URLClassLoader.doLoadClass(name: String, resolve: Boolean)

```
1: if isBarrierClass(name) then
2:     defineClass(name, bytes, 0, bytes.length)
3: else if !isSharedClass(name) then
4:     super.loadClass(name, resolve)
5: else
6:     baseClassLoader.loadClass(name)
7: end if
```

创建 HiveClient 的具体实现逻辑（删除了异常处理的代码）如以下代码所示。当 isolationOn 为 false 时，直接创建一个 HiveClientImpl 对象并返回。代码中的 classLoader 为 MutableURLClassLoader 类型，可以直接通过 addURL 方法加载所需要的类。

```
private[hive] def createClient(): HiveClient = {
    if (!isolationOn) {
      return new HiveClientImpl(version, sparkConf, hadoopConf, config, baseClassLoader, this)
    }
    val origLoader = Thread.currentThread().getContextClassLoader
    Thread.currentThread.setContextClassLoader(classLoader)
    classLoader.loadClass(classOf[HiveClientImpl].getName)
        .getConstructors.head
        .newInstance(version, sparkConf, hadoopConf, config, classLoader, this)
        .asInstanceOf[HiveClient]
    Thread.currentThread.setContextClassLoader(origLoader)
}
```

接下来介绍 HiveSessionCatalog。HiveSessionCatalog 中一个重要的方法是查找数据表（lookupRelation）。HiveSessionCatalog 在查找数据表时，正常情况下最终会构造 MetastoreRe-

lation 对象。MetastoreRelation 与默认情况下数据源对应的 LogicalPlan 地位相同，是 Hive 模块下重要的概念，用来表示 Hive 中的数据表。MetastoreRelation 属于 LeafNode 节点类型，同时实现了 MultiInstanceRelation 接口（newInstance 方法）、FileRelation 接口（inputFiles 方法）和 CatalogRelation 接口（catalogTable 与 output 方法），其主要属性和方法如图 10.4 所示。

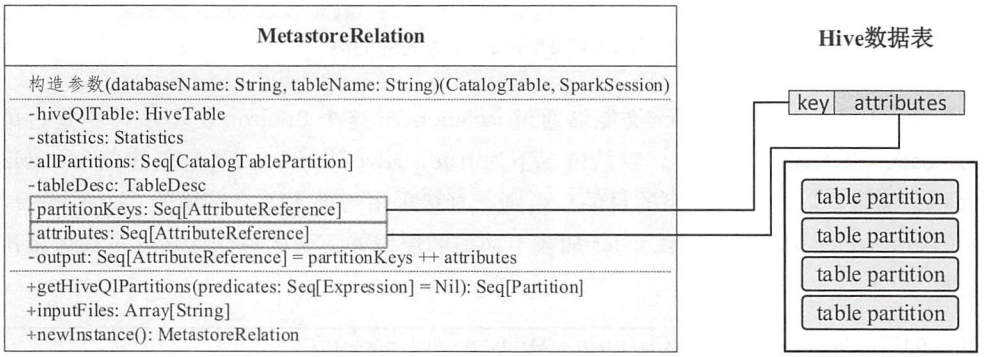


图 10.4　MetastoreRelation 概览

当构造 MetastoreRelation 对象时，主要传入数据库名和数据表名，以及 Spark 中的 SparkSession 与 CatalogTable 对象。图 10.4 中展示了 MetastoreRelation 中比较常用的一些属性。首先是 hiveQlTable 属性，它属于 Hive 中的 HiveTable 类型，也是用来获取 Hive 数据表信息的重要渠道，包括统计信息、分区信息和元数据信息等。其次，TableDesc 类型的 tableDesc 属性也比较重要，当 Spark SQL 系统中需要知道 Hive 数据表存储格式和序列化/反序列化方式时，TableDesc 类型的属性必不可少。此外，需要注意的是，MetastoreRelation 中采用 Spark 的 CatalogTable 定义方式，output 由数据行属性字段（attributes）和分区字段（partitionKeys）组合而成；而在某些 Hive 版本中，分区字段是包含在数据行属性字段中的。除这些常用的属性外，MetastoreRelation 还实现了对应接口的方法，并对一些关键的工具函数（如 equals、hashcode 等）进行了重载。

10.2.2 Analyzer 之 Hive-Specific 分析规则

在逻辑计划阶段，HiveSessionState 中仍然会生成新的 Analyzer，如图 10.5 所示，可以看到，不同之处在于 extendedCheckRules 中少了 HiveOnlyCheck 规则，且 extendedResolutionRules 中多了 ParquetConversions 和 OrcConversions 两条规则。在默认的 Analyzer 中，HiveOnlyCheck 规则会遍历逻辑算子树，如果发现 CreateTable 类型的节点且对应的 CatalogTable 是 Hive 才能够提供的，则会抛出 AnalysisException 异常，因此在 Hive 场景下，这条规则不再需要。

顾名思义，ParquetConversions 与 OrcConversions 用来处理 Parquet 数据表文件与 ORCFile 数据表文件。前面已经分析过，在 Hive 模块中，数据表统一用 MetastoreRelation 表示，而

MetastoreRelation 包含了复杂的 partition 信息。当一个查询涉及的数据表不涉及分区情况时，为了得到更优的性能，可以将 MetastoreRelation 直接转换为数据源表（Data source table）。具体来讲，包含两种情况。

- 读数据表，将 LogicalPlan 中所有满足条件的 MetastoreRelation 转换为 Parquet（ORCFile）文件格式所对应的 LogicalRelation 节点。
- 写数据表，即 InsertIntoTable 逻辑算子节点，同样的逻辑替换目标数据表 MetastoreRelation 为对应的 LogicalRelation 节点。具体的实现可以参见 convertToLogicalRelation 方法。

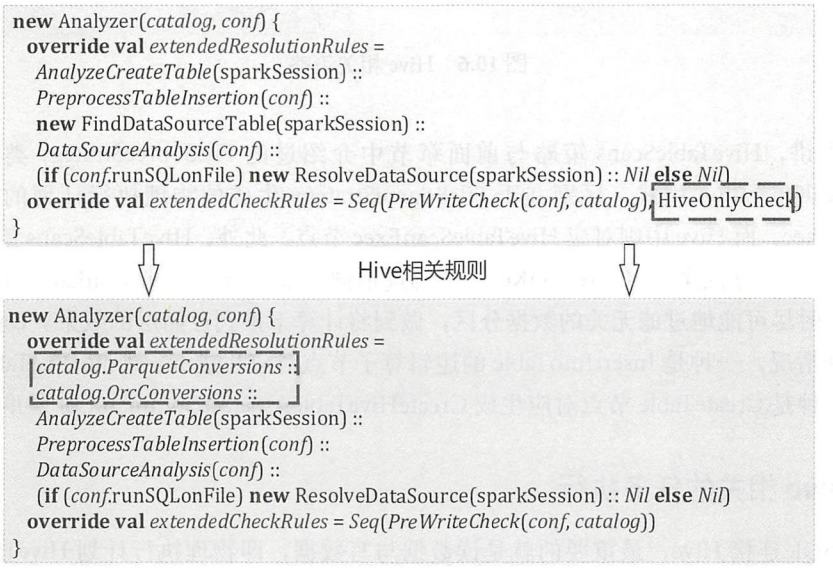


图 10.5 Hive 相关分析规则

10.2.3 SparkPlanner 之 Hive-Specific 转换策略

在物理计划阶段，HiveSessionState 中创建的 SparkPlanner 与 SessionState 中默认创建的 SparkPlanner 的区别如图 10.6 所示。由此可见，在 Hive 场景中，extraStrategies 中多了 HiveTableScans、DataSinks 和 Scritps 这 3 条策略。

这 3 条策略封装在 HiveStrategies 接口中，其含义也比较好理解，HiveTableScans 策略负责从 Hive 表中读取数据，DataSinks 策略用于向 Hive 表中写入数据，而 Scripts 则是应用于 SQL 脚本的场景（比较少见，这里不进行展开）。因此，Hive 场景下 Spark SQL 中的物理计划的各种策略仍然会复用，HiveTableScans 策略与 DataSinks 策略就像一头一尾，形成了对 Hive 中数据处理的闭环。

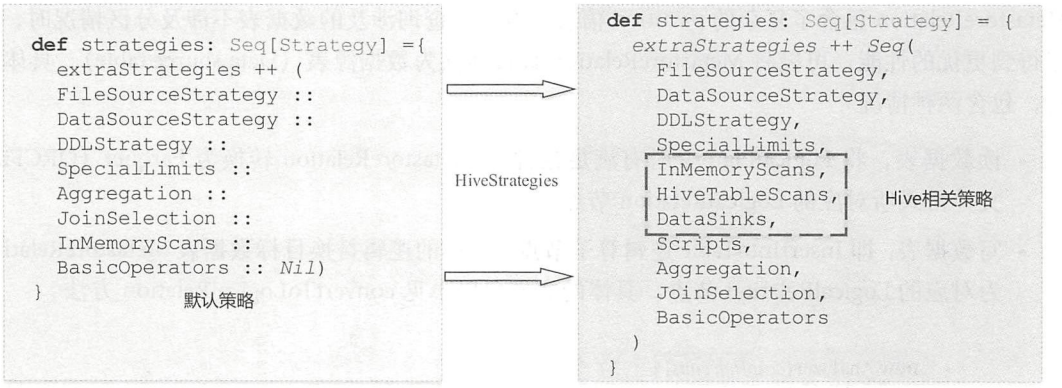


图 10.6　Hive 相关策略

具体来讲，HiveTableScans 策略与前面章节中介绍过的 FileSourceStrategy 类似，同样是匹配 PhysicalOperation 模式，区别在于 FileSourceStrategy 生成的物理执行计划的节点为 FileSourceScanExec，而 Hive 中则对应 HiveTableScanExec 节点。此外，HiveTableScans 策略会将涉及 MetastoreRelation 分区属性（partitionKey）的筛选谓词下推到 HiveTableScanExec 的构造参数中，用于在执行时尽可能地过滤无关的数据分区，做到将计算下推到存储层的效果。DataSinks 策略会考虑两种情况，一种是 InsertIntoTable 的逻辑算子节点对应生成 InsertIntoHiveTable 物理执行节点，另一种是 CreateTable 节点对应生成 CreateHiveTableAsSelectCommand 命令并执行。

10.2.4 Hive 相关的任务执行

Spark SQL 连接 Hive，最重要的就是读数据与写数据，即物理执行计划 HiveTableScanExec 与 InsertIntoHiveTable，本节详细介绍这两个类的实现。

负责读数据的 HiveTableScanExec 的实现相对简单，作为叶子节点，读者很容易想到需要生成 HadoopRDD 来处理输入数据。HiveTableScanExec 的构造参数中比较重要的是代表 Hive 数据表的 relation（类型为 MetastoreRelation）和代表 HivePartition 相关的谓词 partitionPruningPred（类型为 Seq[Expression]）。因此，根据 MetastoreRelation 中是否包含数据分区，具体分为两种情况：如果 relation 中没有数据分区的定义，则直接调用 makeRDDForTable 方法；如果 relation 中定义了数据分区，则调用 makeRDDForPartitionedTable 方法。这两个方法的具体实现由 HadoopTableReader 完成，如图 10.7 所示。在 HadoopTableReader 的实现中，根据 relation 所在的 HDFS 路径创建 HadoopRDD 后，还需要支持将 Hive 数据表中的 Writable 数据行转换为常规的 Row 类型（参见 fillObject 方法）。

当写入数据到 Hive 中时，InsertIntoHiveTable 节点不再产生新的数据，具体的操作过程在 sideEffectResult 的计算中进行（最终返回的是 Seq.empty[InternalRow]）。doExecute 方法中

调用的是 sparkContext.parallelize，可将并行度设置为 1。数据写入到 Hive 数据表的逻辑最终落在 saveAsHiveFile 方法中，对于 InsertIntoHiveTable 的子节点得到的 RDD，启动 Spark 执行 sparkContext.runJob 方法提交任务。图 10.7 中的 SparkHiveWriterContainer 与 SparkHiveDynamicPartitionWriterContainer 实际上是将 RDD 数据写入 Hive 数据表和分区的对象。

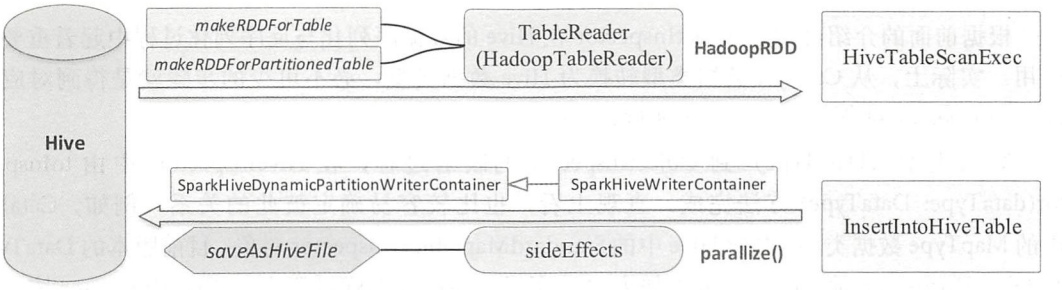


图 10.7 Spark SQL 中读写 Hive 数据的执行过程

10.3 Spark SQL 与 Hive 数据类型

当不同系统之间涉及数据交互时，数据类型的映射和转换是一个不可避免的问题。同样的，在 Spark SQL 连接 Hive 的场景下，Catalyst 中定义的数据类型与 Hive 中的数据类型间的互相转换是一个重要的需求。

在 Spark SQL 的 Hive 模块下，由 HiveInspectors 负责 Catalyst 与 Hive 数据类型的转换，具体包括 4 个方面的功能。

- 数据解封（Data Unwrapping），将 Hive 中的数据类型转换为 Catalyst 中的数据类型，称为 unwrap 操作。
- 数据封装（Data Wrapping），将 Catalyst 中的数据类型转换为 Hive 中的数据类型，称为 wrap 操作。
- 将 Catalyst 中的数据绑定相应的 ObjectInspector 对象，对应 toInspector 操作。
- 从 ObjectInspector 对象中提取 Catalyst 数据，对应 inspectorToDataType 操作。

10.3.1 Hive 数据类型与 SerDe 框架

Hive 中的数据类型比较完善，不仅支持基本的数据类型，还支持集合数据类型（如 Map、Struct 和 Array 等）。SerDe 是 Hive 中用来实现数据序列化和反序列化的框架，SerDe 层构建在数据存储和执行引擎之间，能够实现数据存储与执行引擎之间的解耦。SerDe 体系中提供了一

个辅助类 ObjectInspector，可帮助使用者访问需要序列化或反序列化的对象。具体内容这里不再继续展开，有兴趣的读者可以参考官方文档。

10.3.2 DataTypeToInspector 与 Data Wrapping

根据前面的介绍可知，ObjectInspector 在 Hive 的数据序列化与反序列化过程中起着重要的作用。实际上，从 Catalyst 数据类型转换为 Hive 数据类型，必不可少的步骤就是得到对应的 ObjectInspector 对象，然后实现数据封装操作。

数据类型（DataType）到 ObjectInspector 的映射过程，在 HiveInspectors 中由 toInspector(dataType: DataType) 方法完成。直观上看，也比较容易确定彼此的关系。例如，Catalyst 中的 MapType 数据类型，对应 Hive 中的 StandardMapObjectInspector 对象。目前版本的 DataType 与 ObjectInspector 对象的对应关系如表 10.2 所示。在模式匹配过程中，ObjectInspector 统一从 Hive 的 ObjectInspectorFactory 工厂类获得。

表 10.2　Catalyst 数据类型对应的 Hive ObjectInspector

DataType	Hive ObjectInspector
ArrayType	ObjectInspectorFactory.getStandardListObjectInspector
MapType	ObjectInspectorFactory.getStandardMapObjectInspector
StringType	PrimitiveObjectInspectorFactory.javaStringObjectInspector
IntegerType	PrimitiveObjectInspectorFactory.javaIntObjectInspector
DoubleType	PrimitiveObjectInspectorFactory.javaDoubleObjectInspector
BooleanType	PrimitiveObjectInspectorFactory.javaBooleanObjectInspector
LongType	PrimitiveObjectInspectorFactory.javaLongObjectInspector
FloatType	PrimitiveObjectInspectorFactory.javaFloatObjectInspector
ShortType	PrimitiveObjectInspectorFactory.javaShortObjectInspector
ByteType	PrimitiveObjectInspectorFactory.javaByteObjectInspector
NullType	PrimitiveObjectInspectorFactory.javaVoidObjectInspector
BinaryType	PrimitiveObjectInspectorFactory.javaByteArrayObjectInspector
DateType	PrimitiveObjectInspectorFactory.javaDateObjectInspector
TimestampType	PrimitiveObjectInspectorFactory.javaTimestampObjectInspector
DecimalType()	PrimitiveObjectInspectorFactory.javaHiveDecimalObjectInspector
StructType(fields)	ObjectInspectorFactory.getStandardStructObjectInspector

除支持直接从 DataType 得到 ObjectInspector 对象外，HiveInspectors 还实现了 toInspector(expr: Expression) 方法，支持从 Expression 得到 ObjectInspector 对象，对应 3 种不同的情况。

(1) Literal 类型的表达式，例如 expr 为 Literal（value, DoubleType），那么可以确定数据类型为 DoubleType。

(2) 可折叠（foldable）表达式，用 expr 执行后的结果创建 Literal，然后重复情况 (1)。

(3) 非常量表达式，根据 expr.dataType 进行判断。

当有了 DataType 与 ObjectInspector 之后，就可以得到数据封装的函数了。在 HiveInspectors 中具体实现为 wrapperFor(oi: ObjectInspector, dataType: DataType) 方法，返回类型为“Any => Any”的函数。wrapperFor 根据 ObjectInspector 的类型与 DataType 的类型进行相应处理，例如，当 oi 为 StructObjectInspector 对象时（代码如下），首先将 dataType 转换为具体的 StructType 类型，然后针对 StructType 中每个字段得到封装函数，最终得到的是一个封装函数数组，其中 withNullSafe 用来添加逻辑单独处理 null 的情况。

```
case soi: StructObjectInspector =>
      val structType = dataType.asInstanceOf[StructType]
      val wrappers = soi.getAllStructFieldRefs.asScala.zip(structType).map {
        case (ref, tpe) => wrapperFor(ref.getFieldObjectInspector, tpe.dataType)
      }
      withNullSafe { o =>
        val row = o.asInstanceOf[InternalRow]
        val result = new java.util.ArrayList[AnyRef](wrappers.size)
        soi.getAllStructFieldRefs.asScala.zip(wrappers).zipWithIndex.foreach {
          case ((field, wrapper), i) =>
          val tpe = structType(i).dataType
          result.add(wrapper(row.get(i, tpe)).asInstanceOf[AnyRef])
        }
        result
```

基于 wrapperFor 进行 Spark SQL 的数据封装就很容易了，可用 wrap(a: Any, oi: ObjectInspector, dataType: DataType) 方法完成，直接根据 oi 与 dataType 得到对应的封装函数，然后将 a 作为封装函数的输入即可。

10.3.3 InspectorToDataType 与 Data Unwrapping

从 Hive 数据转换为 Spark SQL 中的数据对应数据的解封装，具体有两种操作，第一种是能够根据 Hive 中的 ObjectInspector 确定 Spark SQL 中的数据类型，第二种是能够根据 Hive 中的 ObjectInspector 得到数据解封装函数。

从 ObjectInspector 确定 DataType 的逻辑很简单，直接枚举所有情况即可，具体实现参见 inspectorToDataType 方法。ObjectInspector 与 DataType 的映射关系如表 10.3 所示。可见，一种 DataType 一般对应两种 ObjectInspector 对象，例如 FloatType 对应 WritableFloatObjectInspector 和 JavaFloatObjectInspector 对象。

数据的解封装操作本质上是对 ObjectInspector 对象的解封装操作，在 HiveInspectors 中实现为 unwrapperFor(objectInspector: ObjectInspector) 方法，返回类型同样是“Any => Any”的函数。该方法严格遵循以下顺序。

表 10.3　Hive ObjectInspector 与 DataType 的映射关系

Hive ObjectInspector	DataType
StructObjectInspector	StructType
ListObjectInspector	ArrayType
MapObjectInspector	MapType
WritableStringObjectInspector	StringType
JavaStringObjectInspector	StringType
WritableHiveVarcharObjectInspector	StringType
JavaHiveVarcharObjectInspector	StringType
WritableHiveCharObjectInspector	StringType
JavaHiveCharObjectInspector	StringType
WritableIntObjectInspector	IntegerType
JavaIntObjectInspector	IntegerType
WritableDoubleObjectInspector	DoubleType
JavaDoubleObjectInspector	DoubleType
WritableBooleanObjectInspector	BooleanType
JavaBooleanObjectInspector	BooleanType
WritableLongObjectInspector	LongType
JavaLongObjectInspector	LongType
WritableShortObjectInspector	ShortType
JavaShortObjectInspector	ShortType
WritableByteObjectInspector	ByteType
JavaByteObjectInspector	ByteType
WritableFloatObjectInspector	FloatType
JavaFloatObjectInspector	FloatType
WritableBinaryObjectInspector	BinaryType
JavaBinaryObjectInspector	BinaryType
WritableHiveDecimalObjectInspector	decimalTypeInfoToCatalyst
JavaHiveDecimalObjectInspector	decimalTypeInfoToCatalyst
WritableDateObjectInspector	DateType
JavaDateObjectInspector	DateType
WritableTimestampObjectInspector	TimestampType
JavaTimestampObjectInspector	TimestampType
WritableVoidObjectInspector	NullType
JavaVoidObjectInspector	NullType

（1）常量（ConstantObjectInspector）且为 null，返回 nul。

（2）常量（ConstantObjectInspector），根据 ObjectInspector 提取其中的数据。

（3）如果 ObjectInspector 倾向于获取 Writable 类型（preferWritable），那么首先得到 Writable 类型的数据，然后转换为 Catalyst 中的数据，否则直接由 ObjectInspector 得到 Java 数据类型。

总的来讲，目前 Spark SQL 与 Hive 之间的数据映射转换在功能上除不支持 Hive 中的 HiveVarchar/HiveChar 类型和 Union 类型外，基本能够满足各种场景的需求。从代码实现层面来看，后续的 Spark 版本规划中将会优化整体实现方式，以及一些特殊情形的处理。例如，函数 date_add(printf("%s-%s-%s", a,b,c), 3) 就不需要进行数据的 unwrapping 和 wrapping 操作。

10.4 Hive UDF 管理机制

Spark SQL 连接 Hive 时，除数据类型需要处理外，另一个比较重要的部分就是 Hive 中的 UDF、UDAF 和 UDTF 等用户自定义函数的管理。

Hive UDF、UDAF 和 UDTF 与 Spark SQL 表达式的对应关系如图 10.8 所示。Hive 中的 UDF 对应 HiveSimpleUDF 对象，HiveSimpleUDF 是 Expression 的子类；Hive 中的 GenericUDF 对应 HiveGenericUDF 对象，HiveGenericUDF 也是 Expression 的子类；Hive 中的 AbstractGenericUDAFResolver 与 UDAF 均对应 HiveUDAFFunction 对象，HiveUDAFFunction 继承自 ImperativeAggregate 类，实现了相关的聚合操作接口；Hive 中的 GenericUDTF 对应 HiveGenericUDTF 对象，而 HiveGenericUDTF 是 Generator 的子类。需要注意的是，HiveGenericUDTF 中不会维护输入数据行与行之间的状态，因此在 UDTF 中输出结果之前，分区动作相关的操作（例如 close 调用）会存在与 Hive 中不一致的情况。虽然在实践中发现这种情况不会影响彼此之间的兼容性，但是如果对输入数据之间的状态有要求，就可以实现为自定义的聚合函数。

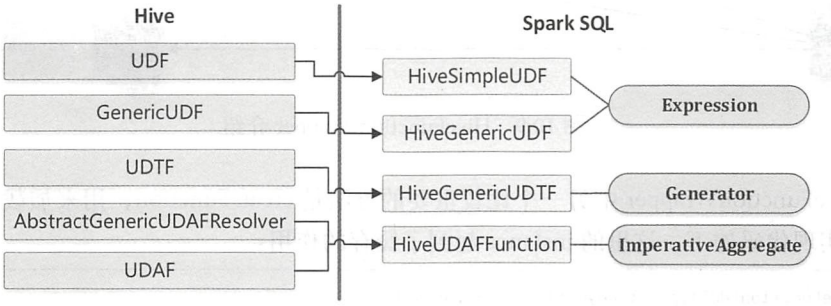


图 10.8 Hive UDF、UDAF 和 UDTF 与 Spark SQL 表达式的对应关系

在 HiveSessionCatalog 类中重载了 makeFunctionBuilder 方法。以 HiveSimpleUDF 为例，当判定对应的类为 Hive 中的 UDF 类型时，就会创建 HiveSimpleUDF 对象。构造的参数有 3 个，分别是函数名（name）、根据 UDF 类 clazz 得到的 HiveFunctionWrapper（clazz.getName）和 UDF 输入参数（children: Seq[Expression]）。注意这里的 HiveFunctionWrapper，它实现了对 Hive 中 UDF 函数的封装，在 Spark SQL 的 Hive UDF 管理中起着重要作用。

为何要用 FunctionWrapper？它和 Hive 不同版本 UDF 的实现机制有关。Hive 0.12.0 版本及其之前的版本，都可以直接在 Executor 节点上创建一个新的 UDF 对象，而从 Hive 0.13.1 版本开始，UDF 对象需要在 Driver 节点上创建完成初始化，然后进行序列化，最后在 Executor 节点上执行反序列化操作。然而，并非所有的对象都是可序列化的，例如，GenericUDF 类就没有实现 Serializable 接口。在 Hive 中，为解决这个问题，引入了 Kryo 和 XML 序列化的方式。相应的，

在 Spark SQL 中，FunctionWrapper 负责完成这部分功能。

如图 10.9 所示，HiveFunctionWrapper 中主要实现了 3 个方面的接口。其中，writeExternal 与 readExternal 方法对应 Externalizable 接口，用于 Driver 端序列化与 Executor 反序列化操作。基于性能的考虑，序列化与反序列化功能的实现使用的是 Kryo 框架。需要注意的是，由于实现层面的考虑，这里的 Kryo 直接采用 Hive 中的 Utilities.runtimeSerializationKryo，而并非 Spark 本身的。

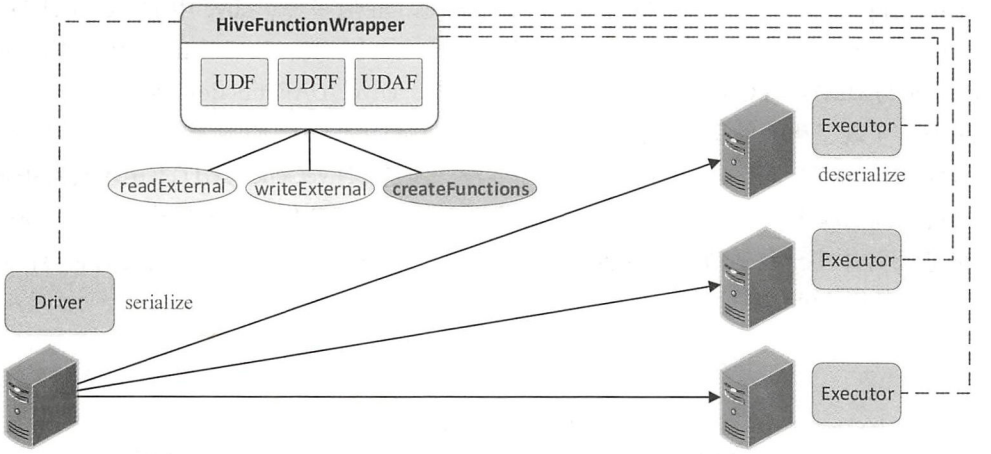


图 10.9　HiveFunctionWrapper 介绍

在 HiveFunctionWrapper 中另一个比较重要的方法是 createFunction，用来加载 UDF 相应的类，具体实现代码如下。这里的 instance 起到了缓存的作用。

```
def createFunction[UDFType <: AnyRef](): UDFType = {
    if (instance != null) {
      instance.asInstanceOf[UDFType]
    } else {
      val func = Utils.getContextOrSparkClassLoader.loadClass(functionClassName).newInstance.
    asInstanceOf[UDFType]
      if (!func.isInstanceOf[UDF]) {
        instance = func
      }
      func
    }
}
```

CreateFunction 方法在 HiveSimpleUDF、HiveGenericUDF、HiveGenericUDTF 和 HiveUDAFFunction 类中都用到了。以 HiveSimpleUDF 为例，通过 createFunction[UDF]() 得到对应的 function 对象。在 HiveSimpleUDF 执行的方法 (eval) 中，直接通过 Hive 中的 FunctionRegistry 调用 function 处理传入的数据。

10.5 Spark Thrift Server 实现

Thrift [49] 是一种接口描述语言和二进制通信协议，由 Facebook 开发并贡献到 Apache 开源社区，用来定义和创建跨语言的服务。Thrift 包含的代码生成引擎可以应用于多种语言中，包括 C++、Java、Python 等。其数据传输采用二进制格式，相对常用的 XML 和 JSON 格式体积更小，在多语言、高并发和大数据场景下更具优势。

Thrift 框架支持使用 IDL（Interface Definition Language）定义服务接口，然后利用提供的编译器将服务接口编译成不同语言的实现代码，从而实现服务端和客户端跨语言的支持。Spark Thrift Server 中定义的 Thrift 协议在 if 目录下的 TCLIService.thrift 文件中。客户端与服务端工作的原理如图 10.10 所示，协议层（Protocol）、传输层（Transport）乃至底层 IO 传输的具体实现都不需要用户关心。

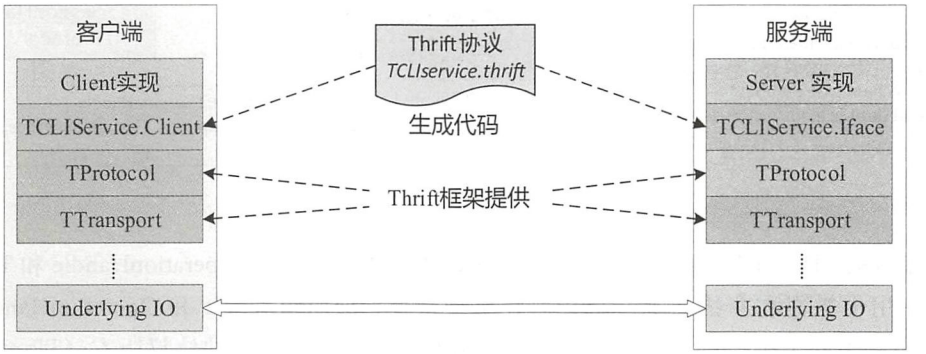


图 10.10 Thrift 协议与代码生成

Thrift 框架根据 TCLIService 协议生成的代码位于 apache.hive.service.cli.thrift 目录下，生成源码中的类与 Thrift 协议中定义的元素基本上相对应。例如，Thrift 协议中定义了 Enum 类型的协议版本（TProtocolVersion），对应生成的 TProtocolVersion 类也是枚举类型（继承 Thrift 框架中的 TEnum 类）。Thrift 协议以数据类型定义为主，定义的服务为 TCLIService（以 service 开头），也是客户端和服务端交互的接口。如图 10.10 所示，客户端和服务端只要实现 TCLIService 中相关的接口即可。

当然，在一个完整的系统中，Thrift 生成的代码一般不会直接暴露给上层应用，而是会结合其他方面的需求（如权限管理等），从架构上进行多层的封装。为了方便后续的原理分析，本节首先介绍一些从协议定义到上层封装的典型例子。

1. 句柄（THandleIdentifier/Handle/OperationHandle/SessionHandle）

在客户端与服务端的交互过程中会涉及大量的会话（Session）和操作（Operation），对这些对象的统一管理少不了唯一的标识，类似指针和句柄的概念。针对该需求，TCLIService 协议中定义了 THandleIdentifier 类型，其中包含一个公开的 ID 和一个私密的 ID（都是 Binary 类型），生成的 THandleIdentifier 类对应了 UUID 类型的 pulicId 和 secretId。在此基础上，上层又定义了 Handle 抽象类，其内部包含一个 THandleIdentifier 对象，实现对其的封装，如图 10.11 所示。

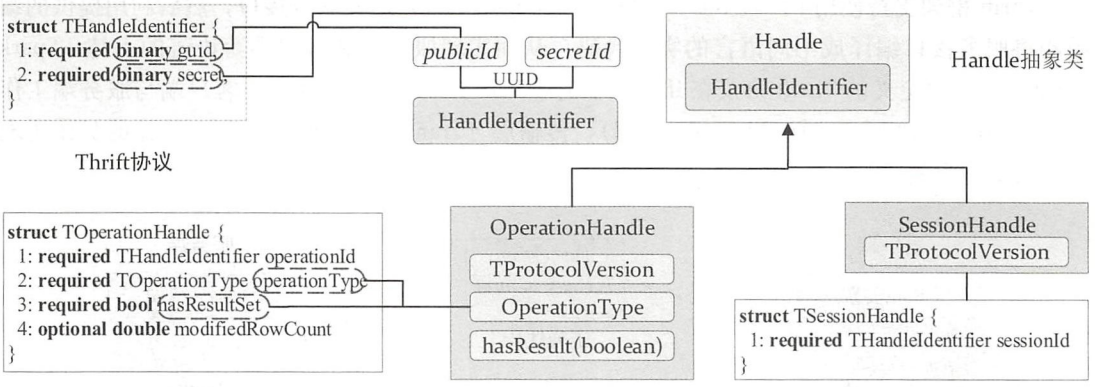


图 10.11　Handle 相关定义与封装

除基本的 THandleIdentifier 外，TCLIService 协议中还定义了 TOperationHandle 和 TSessionHandle，用于操作和会话的唯一标识。在实现层面，SessionHandle 和 OperationHandle 都是 Handle 抽象类的子类。SessionHandle 中额外加入了对应会话采用的协议版本（TProtocolVersion），OperationHandle 中除协议版本外，还加入了操作类型（OperationType）和 hasResult 属性。各种 Handle 对象一般都会应用在 Map 类型的数据结构中，来索引相对应的对象。

2. 数据抓取相关（TFetchOrientation/FetchOrientation/FetchType）

服务端完成操作之后（例如计算结果写入 HDFS 等），客户端会抓取结果数据。TCLIService 协议将数据抓取动作的几种操作定义在 TFetchOrientation 类型中。TFetchOrientation 列举了 6 种操作方式，分别为 FETCH_NEXT（获取下一个数据行集合）、FETCH_PRIOR（获取前一个数据行集合）、FETCH_RELATIVE（根据相对的偏移量获取相关数据行集合）、FETCH_ABSOLUTE（根据绝对的偏移量获取相关数据行集合）、FETCH_FIRST（获取结果集中的第一个数据行集合）和 FETCH_LAST（获取结果集中的最后一个数据行集合）。目前版本的 Spark Thrift Server 实现中仅支持 FETCH_NEXT 和 FETCH_FIRST 两种方式。

在代码层面，Thrift 生成的 TFetchOrientation 类也是 Enum 类型，但最终实际应用的则是

FetchOrientation 类。FetchOrientation 类仅仅是对 TFetchOrientation 类的封装，不做任何额外的操作。此外，FetchType 也是 Enum 类型，表示获取的数据来自查询结果（QUERY_OUTPUT）还是日志（LOG）。

作为服务化的应用，离不开服务端（Server）、客户端（Client）、鉴权管理（Auth）、客户端与服务端会话（Session）和操作（Operation）等基本概念。一般来讲，当服务端（Server）开始接受来自客户端（Client）的请求时，会通过会话管理器（SessionManager）管理维护一系列的会话（Session）信息，SessionManager 又会通过操作管理器（OperationManager）管理一系列的操作（Operation）。Spark Thrift Server 源自 HiveServer2，在实现方面，从上至下进行了多层的封装，如果直接从 Spark 层面作为切入点难度比较大。下面介绍这些基本的概念，然后定位到 Spark 层面的封装和调用。

10.5.1 Service 体系

在 Spark Thrift Server 中，服务端的各个主要模块都实现了 Service 接口来实现统一的管理。整个 Service 体系如图 10.12 所示。Service 的状态有 4 种：创建成功但未初始化（NOTINITED）、初始化完成但还没有开始或结束（INITED）、已经开始（STARTED）和结束（STOPPED）。从图 10.12 中可以看到，Service 接口中定义了 9 个方法，所有的 Service 在初始化时都需要传递 HiveConf 配置信息，这些方法中提供了 register 和 unregister 向该 Service 注册和注销监听器（ServiceStateChangeListener）的功能。

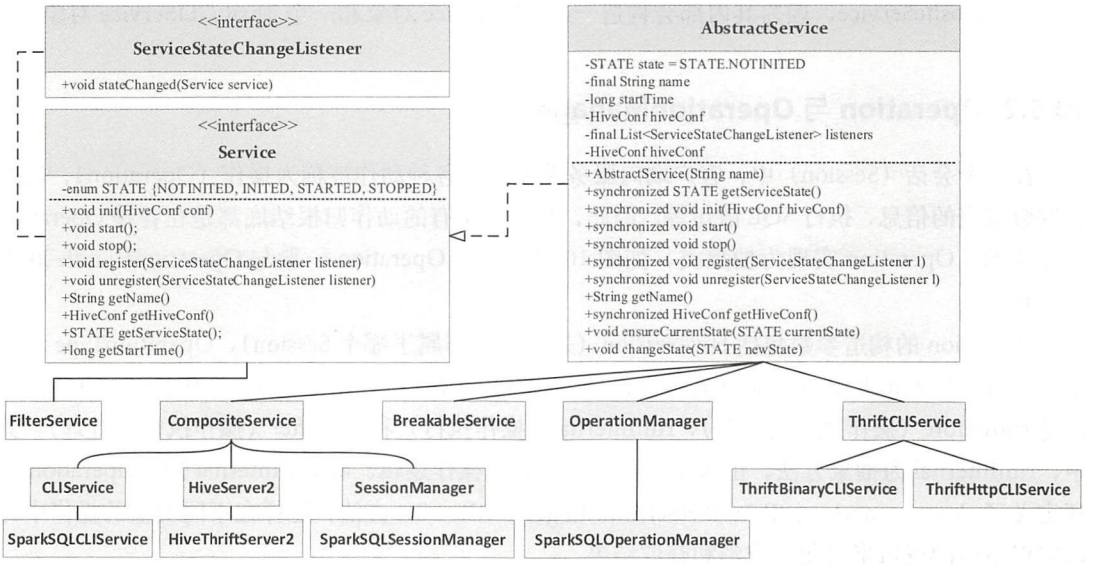


图 10.12 Service 体系概览

AbstractService 是实现了部分 Service 方法的抽象类，默认的状态为 NOTINITED，内部包含了 Service 名（name）和启动时间（startTime）等基本信息，List 类型的 listeners 保存了该 Service 所有的监听器。AbstractService 的操作主要是调用 ServiceOperations 类中提供的静态方法，包括初始化 Service、启动 Service 和停止 Service 等。需要注意的是，AbstractService 实现的大部分方法中都使用 synchronized 关键字完成同步，以支持多线程环境下的操作。除 AbstractService 外，FilterService 也直接实现了 Service 接口，不过仅仅是对 Service 的一次封装，该类在 Spark Thrift Server 的其他模块中并没有用到。类似的，AbstractService 的子类 BreakableService 也是这样，在代码分析过程中可暂时忽略。

图 10.12 中最下面的 SparkSQLCLIService、HiveThriftServer2、SparkSQLSessionManager 和 SparkSQLOperationManager 都是在 Spark 层面的进一步封装，不影响其他模块的完整性，故本节暂不做分析，将其留在最后一个小节讲解。

AbstractService 的子类 OperationManager 用来管理所有的操作。ThriftCLIService 及其两个子类是 Thrift 服务启动的具体实现（IFace 接口）。CompositeService 代表着“组合服务”。CompositeService 类中保存了一个 Service 列表（List<Service>），支持加入 Service（addService）和移除 Service（removeService），所有的启动（start）、停止（stop）等操作都会作用于该列表的所有 Service，统一进行。

图 10.12 中 CompositeService 的子类彼此之间的关系比较复杂，有必要仔细梳理一下。SessionManager 是一种 CompositeService，因为其内部会构造一个 OperationManager 对象；CLIService 是一个 CompositeService，因为其内部会构造一个 SessionManager 对象；HiveServer2 也是一个 CompositeService，因为其内部会构造一个 CLIService 对象和一个 ThriftCLIService 对象。

10.5.2 Operation 与 OperationManager

在一个会话（Session）中，客户端向服务端发起的各种动作被称为操作（Operation），例如获取数据表的信息、执行 SQL 语句等。因此，服务端所有的动作归根结底都是在各种 Operation 中完成的。Operation 实现为抽象类，如图 10.13 所示，Operation 一般与 OperationHandle 联系在一起。

Operation 的构造参数包括 HiveSession（表示该操作属于哪个 Session）、OperationType 和一个 boolean 值 runInBackground（表示操作是否后台异步执行）。在 Operation 类中比较重要的方法是 runBefore（操作执行前准备）、runInternal（操作执行）和 runAfter（操作执行后处理）。其中，runInternal 为抽象方法，由各个子类实现不同的操作逻辑。除 runInternal 外，Operation 中还定义了 close、cancel、getResultSetSchema 和 getNextRowSet 方法，这样在不同类型的操作中可以实现不同的逻辑来关闭、取消和获取结果。

这里简单介绍 Operation 的一些变量和方法实现。首先是 operationTimeout 和 lastAccessTime

这两个长整型的变量，operationTimeout 用来判断一个 Operation 是否已经超时，lastAccessTime 记录了上次访问的时间。在 Operation 初始化时对应当时的系统时间戳，而 operationTimeout 则是设定的超时阈值，获取的是 HiveConf 中设置的变量（hive.server2.idle.operation.timeout）。如果 operationTimeout 为 0，则代表永不超时。Operation 中支持的 FetchOrientation 集合为 FETCH_NEXT 和 FETCH_FIRST。其中，FETCH_NEXT 为默认的方式。此外，Operation 中还定义了 volatile 修饰的 HiveSQLException 和 Future 类型的变量，用来表示异常和异步操作的返回值处理。

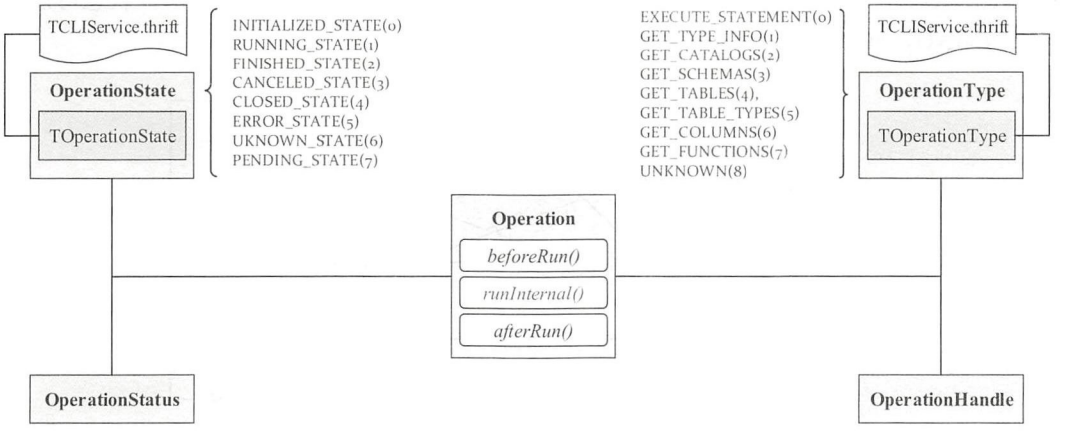


图 10.13 Operation 实现

Operation 作为抽象类，方法以 set/get 为主。除此之外，比较重要的方法是 beforeRun 和 afterRun 中的逻辑。beforeRun 创建了日志文件和相应的 OperationLog 并将其设置到 Hive 的 CurrentOperationLog 中；afterRun 则是将日志文件和相应的 OperationLog 从 CurrentOperationLog 中移除。创建日志文件的核心部分代码如下。可以看到，日志文件存储于 Session 中设置的目录，日志文件名称对应 OperationHandle 中的 HandleIdentifier，同时 OperationLog 对象名称也取决于 OperationHandle 的 toString 方法。

```
protected void createOperationLog() {
    if (parentSession.isOperationLogEnabled()) {
      File operationLogFile = new File(parentSession.getOperationLogSessionDir(),
            opHandle.getHandleIdentifier().toString());
      ..........................................................................
      operationLog = new OperationLog(opHandle.toString(), operationLogFile,
            parentSession.getHiveConf());
      ..........................................................................
      OperationLog.setCurrentOperationLog(operationLog);
    }
  }
```

与 Operation 相关的类主要是 OperationHandle、OperationType、OperationState 和 OperationStatus 4 个。OperationType 是 Enum 类型，本质上是对 Thrift 生成的 TOperationType 的封装，用来标识操作的类型，包括获取所有数据表（“GET_TABLES”）、获取 Schema 信息（“GET_SCHEMAS”）等。OperationState 同样是 Enum 类型，本质上是对 Thrift 生成的 TOperationState 的封装，用来标识操作的当前状态，包括挂起（“PENDING”）、正在运行（“RUNNING”）等。OperationStatus 实现比较简单，在 OperationState 的基础上包含了异常信息（HiveSQLException）。需要注意的是，OperationState 的各种状态是不断变化的，状态与状态之间的转移和自动机类似。OperationState 通过 validateTransition 方法判断转换的合理性，如图 10.14 所示，例如 PENDING 状态可以转换为 RUNNING、FINISHED、ERROR 和 CLOSED 状态。

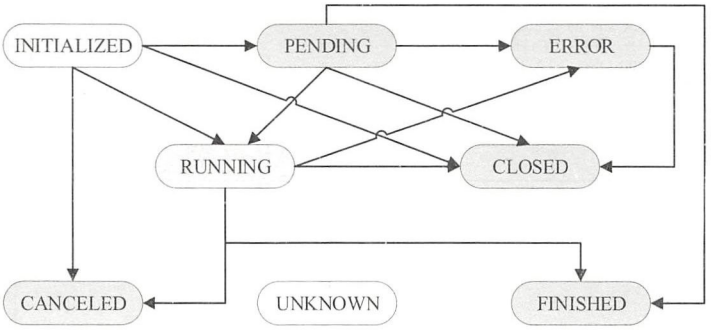


图 **10.14**　Operation 状态转换

Operation 的非抽象子类共有 10 个，如表 10.4 所示。这些具体的操作分为两类，即元数据操作（MetadataOperation）和执行语句操作（ExecuteStatementOperation）。元数据操作共有 7 个，主要用来从 Hive 的元数据库中获取数据表、数据列、相关的 schema 和类型信息。执行语句操作共有 3 个，分别是 Hive 命令操作（HiveCommandOperation）、Hive 中执行 SQL 语句的操作（SQLOperation）和 Spark 中执行 SQL 语句的操作（SparkExecuteStatementOperation）。

表 **10.4**　Operation 的非抽象子类

MetadataOperation	GetCatalogsOperation	获取 catalog 信息
	GetColumnsOperation	获取所有的列
	GetFunctionsOperation	获取所有的函数
	GetSchemasOperation	获取 schema 信息
	GetTableTypesOperation	获取 table 类型信息
	GetTablesOperation	获取所有的表
	GetTypeInfoOperation	获取类型信息
ExecuteStatementOperation	HiveCommandOperation	Hive 命令操作
	SQLOperation	SQL 操作
	SparkExecuteStatementOperation	Spark 执行操作

MetadataOperation 继承自 Operation，是所有元数据操作的父类，在其构造函数中将执行模式设置为非异步（runInBackground 设置为 false）。MetadataOperation 中实现了各种元数据操作的一些共性方法，例如，convertSchemaPattern 方法将 JBDC 中的通配符和转义字符转换为正则表达式中的对应字符，authorizeMetaGets 方法对操作元数据的权限进行验证。ExecuteStatementOperation 实现比较简单，内部创建了一个 HashMap 用来记录查询相关（Query-specific）的配置信息，这些配置信息仅应用于操作本身，不会扩大到 Session 范围。此外，ExecuteStatementOperation 中实现了一个 newExecuteStatementOperation 方法，该方法是静态类型，根据不同情况创建 SQLOperation 对象或 HiveCommandOperation 对象。

这里分析元数据操作中较为简单的 GetSchemasOperation，其他各种 Operation 的具体实现可以参考相应源码。GetSchemasOperation 的核心逻辑实现在 runInternal 中，代码如下，在执行前会将状态设置为 RUNNING。如果 Hive 元数据的 AuthV2 处于开启状态，那么需要对该操作进行权限认证（authorizeMetaGets）。如果满足要求，则 GetSchemasOperation 会从 Session 中获取元数据库的客户端（IMetaStoreClient），根据转换后的 schema 模式，用该客户端获取所有匹配的数据库名，并逐行写入结果集 rowSet 中。如果顺利执行成功，则操作状态相应地设置为 FINISHED；否则操作状态设置为 ERROR，并抛出异常（HiveException）。

```
public void runInternal() throws HiveSQLException {
    setState(OperationState.RUNNING);
    if (isAuthV2Enabled()) {
      String cmdStr = "catalog : " + catalogName + ", schemaPattern : " + schemaName;
      authorizeMetaGets(HiveOperationType.GET_SCHEMAS, null, cmdStr);
    }
    try {
      IMetaStoreClient metastoreClient = getParentSession().getMetaStoreClient();
      String schemaPattern = convertSchemaPattern(schemaName);
      for (String dbName : metastoreClient.getDatabases(schemaPattern)) {
        rowSet.addRow(new Object[] {dbName, DEFAULT_HIVE_CATALOG});
      }
      setState(OperationState.FINISHED);
    } catch (Exception e) {
      setState(OperationState.ERROR);
      throw new HiveSQLException(e);
    }
  }
```

各种 Operation 的管理统一由 OperationManager 来完成，该对象也是与其他模块的接口，其内部维护了一个从 OperationHandle 映射到 Operation 的 HashMap 数据结构。OperationManager 作为 AbstractService 的子类，实现了初始化（init）、启动（start）和停止（stop）的方法。初始化的过程中会将 LogDivertAppender 注册到日志系统中。OperationManager 提供了构造各种 Operation 的功能，例如，newGetCatalogsOperation 方法会返回 GetCatalogsOperation 对象。构造完一个 Operation 之后，会将其加入到内部的 HashMap 中，后续可以通过其 OperationHandle

访问该 Operation 对象。除此之外，针对超时的 Operation 对象，OperationManager 也实现了用 removeExpiredOperations 方法进行批量移除。

10.5.3 Session 与 SessionManager

Session 代表服务器（Server）和客户端（Client）一次会话的过程，直到服务端关闭或客户端关闭时结束。服务端一般会管理多个 Session，每个 Session 对应一个客户端。Spark Thrift Server 中的 Session 定义与实现都在“session”目录下。Session 的功能分为两部分，即 HiveSessionBase 和 HiveSession 两个接口，如图 10.15 所示。顾名思义，HiveSessionBase 中定义的主要是会话的基本内容，如用户名、密码等。类似 Operation 与 OperationHandle 的关系，每个 Session 都对应一个 SessionHandle。此外，Session 的管理统一由 SessionManager 提供。

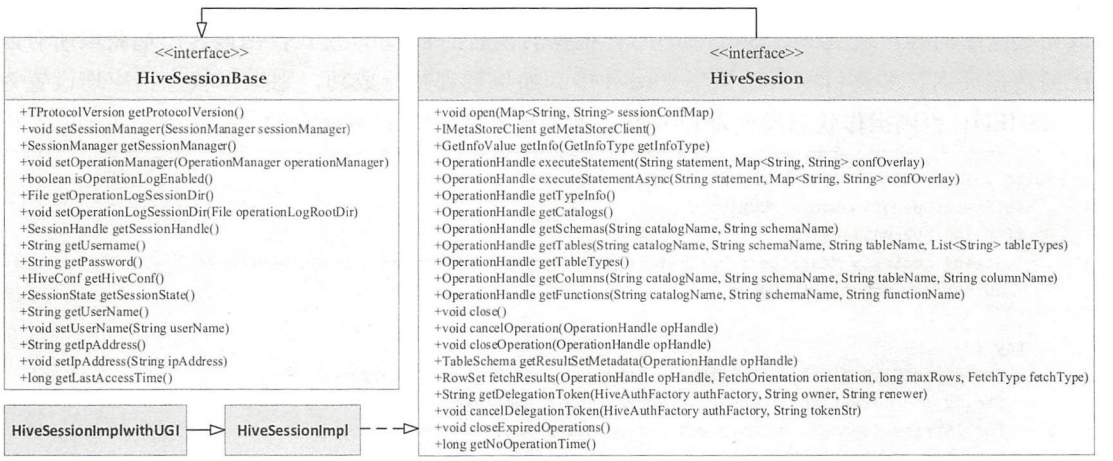


图 10.15　Session 的基本功能

展开来讲，HiveSessionBase 中定义了 17 个接口，可以分为以下几类。

- 客户端与服务端连接采用的 Thrift 协议版本 TProtocolVersion。
- 基本的连接信息，包括用户名 (getUserName/getUsername)、密码 (getPassword)、IP 地址 (getIpAddress) 和最近访问时间 (getLastAccessTime)。
- Hive 配置信息 (getHiveConf) 与 Hive 的 SessionState。
- 与该 Session 相关的模块，包括 SessionManager、OperationManager 和 SessionHandle。
- 与该 Session 中操作日志相关的信息，包括是否支持操作日志（isOperationLogEnabled）和所有操作日志的根目录（getOperationLogSessionDir）。

HiveSession 在 HiveSessionBase 的基础上又提供了 20 多个接口，从逻辑上也可以分为若干类。

- Session 的 open 和 close 操作，其中 open 会传递一个 Map 类型的配置信息作为参数。
- 获取相关的信息 GetInfoType，可以参考 TGetInfoType 中定义的 47 种信息。
- 获取 Hive 元数据库的客户端 IMetaStoreClient，执行元数据操作的基础工作。
- 执行各种操作，例如 getCatalogs，其中 executeStatement 可以同步，也可以异步。
- 操作结果相关，包括获取结果的 TableSchema（getResultSetMetadata）和获取结构（fetchResults）。
- 权限相关，包括 getDelegationToken、cancelDelegationToken 和 renewDelegationToken 的接口。

在图 10.15 中，HiveSession 接口的实现是 HiveSessionImpl 类。构造一个 HiveSessionImpl，只需要 5 个参数：Thrift 的协议版本（TProtocolVersion）、用户名（username）、密码（password）、服务端的 Hive 配置信息（HiveConf）和客户端的 IP 地址（ipAddress）。HiveSessionImpl 中比较重要的是 open 方法，该方法会构造一个 Hive 的 SessionState 对象，代码如下。当服务端在同一个 Session 中执行多条查询语句时，这个 SessionState 会被复用。在 HiveSessionImpl 的实现中，IMetaStoreClient 直接通过 Hive 的静态方法获得。在图 10.15 中也可以看到，HiveSessionImplUGI 是 HiveSessionImpl 的子类，主要特点是加上了 UGI（User Group Information）的管理。

```
public void open(Map<String, String> sessionConfMap) throws HiveSQLException {
  sessionState = new SessionState(hiveConf, username);
  sessionState.setUserIpAddress(ipAddress);
  sessionState.setIsHiveServerQuery(true);
  SessionState.start(sessionState);
  try {
    sessionState.reloadAuxJars();
  } catch (IOException e) {
    String msg = "Failed to load reloadable jar file path: " + e;
    LOG.error(msg, e);
    throw new HiveSQLException(msg, e);
  }
  ....................................................
}
```

最后来看 SessionManager，作为服务端所有 Session 的“管家”，管理着所有 Session 的连接和关闭，还有 Session 中的操作（通过 OperationManager）。SessionManager 是 CompositeService 的子类，包含了多种服务。下面首先介绍 SessionManager 中涉及的一些重要对象。

- handleToSession：类型为 ConcurrentHashMap，记录了所有 SessionHandle 到 HiveSession 之间的映射，每次执行 openSession 操作后都会将得到的 HiveSession 加入到该数据结构中。
- OperationManager：即操作管理器，在 SessionManager 中构造，创建每个 HiveSession 之后，都会调用 setOperationManager 方法完成对 OperationManager 的设置。
- backgroundOperationPool：类型为 ThreadPoolExecutor，用来管理所有后台运行操作的线程池，该线程池的大小通过参数 hive.server2.async.exec.threads 进行设置，默认为 100。

当 SessionManager 开启新的 HiveSession 时会调用 openSession 方法，该方法中需要注意两个参数，即 Boolean 类型的 withImpersonation 和 String 类型的 delegationToken（表示代理 Token）。这两个参数都用来表示与权限相关的功能，当 withImpersonation 为 true 时，delegationToken 一般也不为空，此时创建的对象是 HiveSessionImplwithUGI，并通过 HiveSessionProxy 得到代理对象来调用相关方法（实际上将方法调用封装在 UserGroupInformation 的 doAs 模块中）。

10.5.4 Authentication 安全认证管理

在任何一个企业级应用中，安全都是一个不可回避的话题，而用户的认证管理（Authentication）则是系统安全体系的第一道屏障。顾名思义，认证就是对访问者身份识别和鉴定的行为，即验证用户的身份，判断其是否为合法用户。

Spark Thrift Server 认证模块的实现在 auth 目录下，整个模块的调用接口是 HiveAuthFactory 类，如图 10.16 所示。构造一个 HiveAuthFactory 对象只需要 HiveConf 配置信息，根据 HiveConf 中的参数(hive.server2.authentication 和 hive.server2.transport.mode)得到认证类型(authTypeStr)与传输模式（transportMode)。需要注意的是，如果 authTypeStr 为“KERBEROS”，则 HiveAuthFactory 中会创建一个 HadoopThriftAuthBridge.Server 对象 saslServer，由此可见 Spark Thrift Server 的 Kerberos 认证实现依赖 Hive 中的实现。

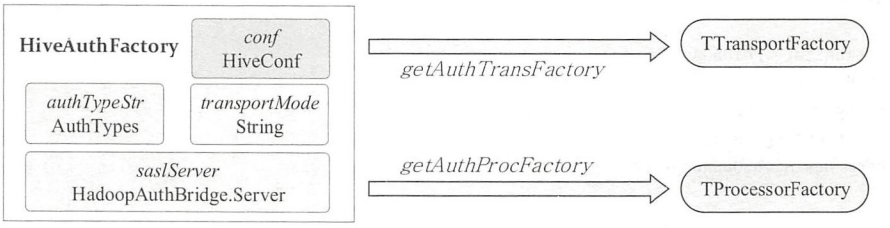


图 10.16　安全认证管理入口 HiveAuthFactory

HiveAuthFactory 中实现了 getAuthTransFactory 和 getAuthProcFactory 两个重要的方法，分别得到 TTransportFactory 和 TProcessorFactory 对象，作为 Thrift Server 启动时必需的元素（参见 ThriftBinaryCLIService 中的实现)。TTransportFactory 对象取决于认证类型，如果是 Kerberos，则由

saslServer 创建，否则统一由 PlainSaslHelper 根据认证类型创建。同样的，TProcessorFactory 的创建也取决于认证类型，Kerberos 方式由 KerberosSaslHelper 创建，其他类型也统一由 PlainSaslHelper 创建。

此外，HiveAuthFactory 中还提供 Kerberos 安全认证机制下涉及 DelegationToken（代理 Token）的一些基本操作（包括 getDelegationToken、cancelDelegationToken 和 renewDelegationToken 等），以及若干的静态工具方法，例如 getSocketTransport 和 getSSLSocket 方法得到的 TTransport 对象等。

当 PlainSaslHelper 构造 TTransportFactory 时，必不可少的构造参数是 Java 中的回调处理对象，即 CallbackHandler（在 Spark Thrift Server 中的实现为 PlainServerCallbackHandler）。PlainServerCallbackHandler 中重载实现的 handle 方法涉及对用户名和密码的验证，由 PasswdAuthenticationProvider 接口提供。在 Spark Thrfit Server 的 auth 模块中，提供用户名和密码认证的实现共有 4 种，如表 10.5 所示。

表 10.5 PasswdAuthenticationProviderr 的不同实现

PasswdAuthenticationProvider	Description
AnonymousAuthenticationProviderImpl	不做任何操作（NONE）
CustomAuthenticationProviderImpl	可配置（CUSTOM）认证实现
LdapAuthenticationProviderImpl	LDAP 认证方式
PamAuthenticationProviderImpl	PAM 实现的认证方式

PasswdAuthenticationProvider 的 4 种实现可以分别简称为 NONE、CUSTOM、LDAP 和 PAM，可以通过工厂类 AuthenticationProviderFactory 获取。NONE 表示什么都不做，即对用户名和密码不进行任何限制；CUSTOM 意味着可定制，由用户实现后通过配置文件加载；LDAP 认证方式依赖 HiveConf 来传递 URL 等相关参数；PAM 认证方式的实现依赖第三方提供的库。

总的来看，Spart Thrift Server 中的安全认证模块对 Hadoop 和 Hive 的依赖较重，特别是 Kerberos 机制的实现。在实现层面，整个模块以 HiveAuthFactory 为主线，同时该类也是其他模块的调用接口，层次上非常清晰。

10.5.5 Spark Thrift Server 执行流程

在前面内容的基础上，本节分析 Spark 中启动 Thrift Server 的主要流程。如图 10.17 所示，整个服务的生命周期从执行 sbin 文件夹下的 start-thriftserver.sh 脚本开始直到执行 stop-thriftserver 脚本结束。

由图 10.17 中的代码可知，Thrift Server 脚本的启动依赖 spark-daemon.sh，最终调用 spark-submit 接口提交 org.apache.spark.sql.hive.thriftserver.HiveThriftServer2 应用。该应用的启动入口在 main 函数中，参看 HiveThriftServer2 的实现可知，main 函数主要完成以下 4 个方面的功能。

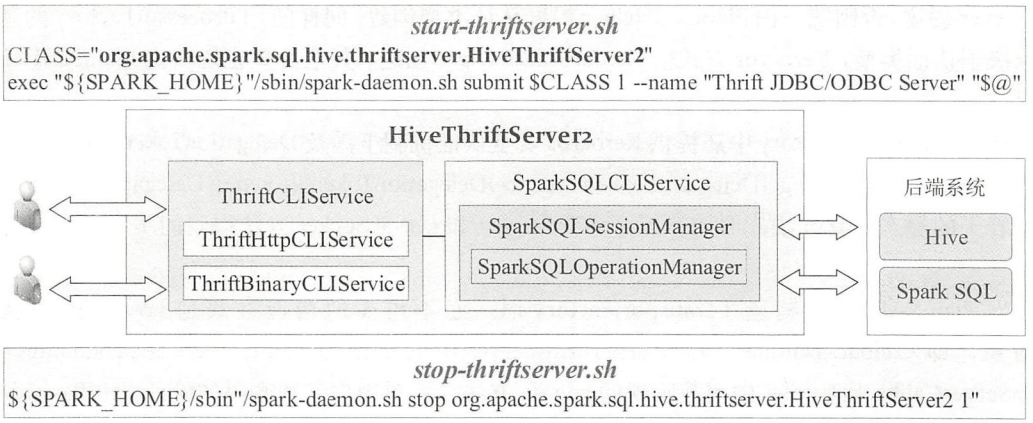


图 **10.17**　Spark SQL Thrift Server 生命周期

- 解析启动参数：构造 HiveServer2 中提供的 ServerOptionsProcessor 对象，解析用户启动服务器时输入的参数（main 函数中的 args）。
- Spark SQL 接口初始化：既然要调用 Spark SQL 执行相关操作，必然少不了 Spark SQL 的相关接口。SparkSQLEnv 中包含 SQLContext 和 SparkContext 两个变量，在 HiveThriftServer2 的 main 函数中调用 init 函数会完成这两个对象的构造。此外，SparkSQLEnv 中的对象在 Spark 关闭时也会回收。
- 构造 Spark SQL 中操作 Hive 的客户端：调用 HiveUtils 中的 newClientForExecution 方法，得到 HiveClient 对象。
- 构造 HiveThriftServer2 对象并启动：将 SparkSQLEnv 中的 SQLContext 对象作为构造参数，构造一个 HiveThriftServer2 对象，随后 HiveThriftServer2 对象采用 HiveClient 中的 HiveConf 完成初始化（Init）并启动（Start）。此外，SparkContext 中会注册一个监听器（HiveThriftServer2Listener）来处理 JobStart 等事件，并添加 ThriftServer 对应的 UI 页面（ThriftServerTab）。

总的来讲，main 函数最终创建一个 HiveThriftServer2 对象实例，作为服务运行于 Spark 集群。HiveThriftServer2 是 HiveServer2 的子类，也是 CompositeService 的一种。在图 10.17 中，HiveThriftServer2 包含 ThriftCLIService 和 SparkSQLCLIService 两个服务，其中 ThriftCLIService 作为服务端负责维护与客户端的连接并将客户端的请求转发至 SparkSQLCLIService，SparkSQLCLIService 通过调用后端 Hive 或 Spark 系统完成运算并把执行结果返回给 ThriftCLIService，最终 ThriftCLIService 把结果返回给客户端。因此，在构造 ThriftCLIService 时也会传入 SparkSQLCLIService 对象，以便处理客户端操作时调用。

ThriftCLIService 有 ThriftHttpCLIService 和 ThriftBinaryCLIService 两种形式，分别对应 Http 传输模式和 Binary 传输模式，通过配置参数(hive.server2.transport.mode)进行判断，默认为 Binary 模式。SparkSQLCLIService 是 CLIService 的子类，构造参数与 CLIService 相比多了一个 SQLContext，其内部主要包含一个 SparkSQLSessionManager 对象，用于 Session 的管理。而 SparkSessionManager 是 SessionManager 的子类，构造参数也比 SessionManager 多了一个 SQLContext，其内部包含一个 SparkSQLOperationManager 对象，用于 Operation 的管理。实际上，SparkSQLCLIService、SparkSQLSessionManager、SparkSQLOperationManager 三者之间的关系类似基本的 CLIService、SessionManager、OperationManager 之间的关系。

SparkSQLSessionManager 初始化流程与 SessionManager 初始化流程相似，其 openSession 和 closeSession 方法都重载了 SessionManager 中对应的方法。其中，比较重要的 openSession 的实现代码如下，首先调用 SessionManager 的 openSession 方法创建一个 Session 对象。

```
def openSession(protocol: TProtocolVersion, username: String, passwd: String, ipAddress: String,
    sessionConf: java.util.Map[String, String], withImpersonation: Boolean, delegationToken:
   String): SessionHandle = {
  val sessionHandle = super.openSession(protocol, username, passwd, ipAddress, sessionConf,
   withImpersonation, delegationToken)
  val session = super.getSession(sessionHandle)
  HiveThriftServer2.listener.onSessionCreated(session.getIpAddress, sessionHandle.getSessionId.
   toString, session.getUsername)
  val sessionState = sqlContext.sessionState.asInstanceOf[HiveSessionState]
  val ctx = if (sessionState.hiveThriftServerSingleSession) {
    sqlContext
  } else {
    sqlContext.newSession()
  }
  ctx.setConf("spark.sql.hive.version", HiveUtils.hiveExecutionVersion)
  if (sessionConf != null && sessionConf.containsKey("use:database")) {
    ctx.sql(s"use ${sessionConf.get("use:database")}")
  }
  sparkSqlOperationManager.sessionToContexts.put(sessionHandle, ctx)
  sessionHandle
}
```

创建 Session 对象之后，会通知监听器（HiveThriftServer2Listener）有 HiveSession 被创建的事件，最后从 SQLContext 中得到 HiveSessionState，设置相关属性（version 和 database），并将 SessionHandle 和 HiveSessionState 加入到 SparkSQLOperationManager 的 sessionToContexts 数据结构中。

实际上，SparkOperationManager 只是重载了 OperationManager 的 newExecuteStatementOperation 方法，代码如下。两者之间的主要区别在于，OperationManager 创建的是 ExecuteStatementOperation，查询由 Hive 执行，而 SparkOperationManager 创建的是 SparkExecuteStatement-

Operation，查询发送给 Spark SQL 完成。

```
override def newExecuteStatementOperation(parentSession: HiveSession, statement: String,
   confOverlay: JMap[String, String], async: Boolean): ExecuteStatementOperation = synchronized {
    val sqlContext = sessionToContexts.get(parentSession.getSessionHandle)
    val sessionState = sqlContext.sessionState.asInstanceOf[HiveSessionState]
    val runInBackground = async && sessionState.hiveThriftServerAsync
    val operation = new SparkExecuteStatementOperation(parentSession, statement, confOverlay,
      runInBackground)(sqlContext, sessionToActivePool)
    handleToOperation.put(operation.getHandle, operation)
    operation
}
```

SparkExecuteStatementOperation 是 Spark SQL 执行 SQL 语句的最终实现，其内部声明了 4 个比较重要的对象：执行 SQL 语句生成的 result（DataFrame 类型）、结果集的迭代器 iter（Iterator[SparkRow] 类型）、结果集头部迭代器 iterHeader（Iterator[SparkRow] 类型）和数据类型 dataTypes（Array[DataType] 类型）。作为 Operation 的子类，外部调用的接口是 runInternal 方法，但其核心逻辑在 execute 方法中实现，代码如下。

```
def execute(): Unit = {
   ..............................
   result = sqlContext.sql(statement)
   ..............................
  iter = {
      val useIncrementalCollect = sqlContext.getConf(
         "spark.sql.thriftServer.incrementalCollect", "false").toBoolean
      if (useIncrementalCollect) {
        result.toLocalIterator.asScala
      } else {
        result.collect().iterator
      }
  }
  val (itra, itrb) = iter.duplicate
  iterHeader = itra
  iter = itrb
  dataTypes = result.queryExecution.analyzed.output.map(_.dataType).toArray
  ..............................
}
```

可以看到，SQL 的执行非常简单，调用 SQLContext 中的 sql() 方法即可，经过逻辑计划阶段和物理计划阶段，最终得到的是 DataFrame 数据类型。如果需要支持增量获取数据的操作，iter 会对应 result 的 toLocalIterator 方法，将所有数据收集到单一节点上。结果集每一列的数据类型根据 AnalyzedLogicalPlan 确定。

10.6 本章小结

本章首先介绍 Spark SQL 连接 Hive 的整体流程，然后在此基础上分析 4 个方面的内容，包括逻辑计划阶段和物理计划阶段中与 Hive 相关的规则、Hive Catalog 体系，以及 Hive 数据读和写的执行过程；接下来重点关注了两个内容，对比分析了 Spark SQL 和 Hive 中的数据类型，以及 Spark SQL 对 Hive 的 UDF 管理；最后从多个层面剖析了 Spark Thrift Server 的架构和各个模块的实现逻辑。

实际上，Hive 作为 Hadoop 生态系统中数据仓库的事实标准，应用非常广泛。在企业应用中，Spark SQL 连接并兼容 Hive 是一个通用的需求。从发展历程来看，即使从 Spark 2.0 版本开始摆脱了 Hive 的 SQL 编译过程，但是对 Hive 元数据模块（Metastore）和 Catalog 体系的依赖仍然存在。当然，这也和 Hive 推出早、发展时间久、稳定性好等特性有关，企业内部的数据存储方式和元数据管理方式已经形成，后续各种系统（如 Spark SQL 和 Impala 等）不得不在此方式上兼容。针对该问题，相信在 Spark SQL 的后续版本发展中，会有相应的计划。

第11章 Spark SQL 开发与实践

本章的主题为开发和实践，实际上也是对前面章节中 Spark SQL 技术原理分析的进一步扩展，旨在结合腾讯海量数据下的复杂业务需求，为开发人员提供一些实战应用经验。本章首先介绍腾讯 SQL 引擎发展演化的历程，然后重点阐述 Spark SQL 系统在腾讯内部的改造和优化之路。这些改造和优化保证了海量数据和业务压力下系统的高可用和高性能，同时满足了不同业务的复杂需求。本章最后将分享 Spark SQL 系统在实际运营中的一些集群管理和业务优化方面的经验教训。

11.1 腾讯大数据平台（TDW）简介

腾讯大数据平台（TDW）[50] 诞生自 2009 年，迄今为止，整个平台的建设已经近十年。如今，TDW 已经发展成为包含数据接入、存储、计算、可视化和应用的综合性平台，图 11.1 中展示了平台各个层面的主要构成。在数据接入方面，主要有团队自研的 TDBank，支持各种场景、各种类型的数据导入。数据存储层以 HDFS、HBase 和 Ceph 为主，也会有小比例的数据存储在其他类型的系统中（如关系数据库）。集群资源管理依赖腾讯内部自研的 Gaia 系统，相比开源

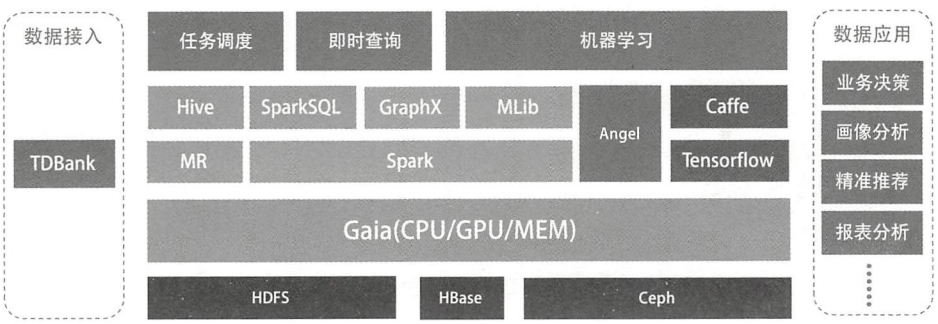


图 11.1　腾讯大数据平台

的 YARN 系统，Gaia 实现了许多扩展特性。在计算层，TDW 不仅改造、融合了各种开源的框架（MapReduce、Spark 等），也针对一些特定的需求，开发自有的系统（例如 Angel）。此外，TDW 在应用层为用户提供针对各种业务场景的前端工具。

如今 TDW 平台承载着日均超过万亿级别的数据采集、PB 级别日均计算量、亿次日均容器数、百 PB 级别的存储规模。它覆盖了腾讯内部七大事业群的各个业务产品，提供丰富的数据应用（业务决策、精准推荐、用户画像、报表分析等）。同时，整个平台也在不断发展壮大，吸收新的技术，发展新的功能，为腾讯各业务产品提供丰富、优质的数据服务。

11.2 腾讯大数据平台 SQL 引擎（TDW-SQL-Engine）

SQL 引擎在腾讯大数据平台中有着承上启下的作用。向下连接 Spark、MapReduce 计算框架，向上满足即时查询和周期性业务。腾讯的 SQL 引擎在开源的基础上，结合腾讯自身业务的特点和需求，开辟出一条拥有自身特色的“道路”。

目前，TDW-SQL-Engine 运行在数万规模的集群节点上，处理日均数百 PB 级别的数据，承载着日均百万级别的 SQL 业务，支撑公司数据业务的交互式分析查询、报表、OLAP 等各种需求，承担在海量数据里挖掘价值的责任。无论是日均业务量，还是处理的数据量，在国内的互联网公司里都位居前列。

11.2.1 SQL-Engine 背景与演化历程

从一开始的 Hive，到 Impala 和 Presto，再到 Spark SQL，业界的 SQL 计算框架在不停地发展，各种可选方案层出不穷。腾讯大数据平台内部的 SQL 引擎（SQL-Engine）也随着业界的发展而不断演进，执行层经历了从 Hive 到 Spark SQL 的更新换代。

在 TDW-SQL-Engine 发展的初期，Hive 是执行层框架的不二选择。实际上，当时业界已经开始采用 Hive 并在实际应用中不断迭代改进。早期的 Hive 存在着许多不足之处，不仅在语法和功能上与用户熟悉的 Oracle 数据库相差太大，而且在开发方面也缺乏友好的可视化界面。此外，Hive 没有提供方便使用的数据仓库与数据应用接口。因此，早期的 Hive 在使用推广上一时难以被用户接受。为了解决这些问题，SQL 引擎团队对 Hive 进行了大量的优化和改造。

- 在功能扩充方面，添加 Oracle 数据库中的常用功能，主要包括：基于角色的权限管理机制，确保数据安全；兼容 Oracle 分区方式，提供二级分区的功能；窗口函数，扩充 Hive 的功能；过程语言多维分析；CTE 表达式的支持；DML 语言扩充；入库数据校验，保证数据质量；与其他数据库的互通。
- 在易用性方面，在 HiveQL 的基础上扩充标准化 SQL 语法，做到业务的无缝迁移，提供非

常友好的可视化集成开发环境。

- 在性能优化方面，采用二进制存储格式，提升读写效率，并支持 LZO 压缩。同时，在扩充 HashJoin、按行划分数据等方面进行了大量技术优化。
- 在稳定性方面，优化和改造主要包括：容灾与负载均衡、大结果集获取接口优化、元数据接口优化、内存泄漏检测、服务过载保护和非线程安全容器优化等。

在 SQL 引擎团队的努力下，优化改造后的 Hive 基本满足了用户需求，也开始逐步支撑腾讯内部绝大部分的离线数据处理业务。

虽然对 Hive 进行了大量优化和改造，但是随着数据量和业务量的增加，Hive 作为 SQL 执行层的性能瓶颈逐渐凸显，一些有着时限要求的应用开始受到影响。Hive 采用 MapReduce 作为计算引擎，而 MapReduce 采用中间结果落盘的机制，大量的磁盘 IO 制约了 SQL 的执行速度。与此同时，Spark 异军突起，构建于其基础之上的 Spark SQL 在执行效率上大幅领先 Hive，作为 SQL 处理领域的新秀发展迅猛，成为业界广泛关注的解决方案。后来，经过初步的测试和版本维护的权衡考虑，团队决定将 SQL 引擎中的执行层从 Hive 逐渐更新换代到 Spark SQL 系统。

整个过程并非一蹴而就，最早引入的 Spark SQL 版本为 1.5，随后，逐步升级到 1.6 版本，再到当前的 2.1 版本（注：2.1 版本的基础上合并了若干 2.2 版本的 patch）。在此过程中，SQL-Engine 团队对 Spark SQL 进行大量的兼容性改造和优化。这些改造和优化包括 400 多个功能特性和性能的优化，涉及数万行的代码改动。这里简单列举其中的一些内容，在后面的内容中会详细介绍。

- 兼容性：兼容大数据平台特有的 SQL 语法（包括 Oracle 数据库）、数据存储格式、分区格式、元数据格式等。
- 新框架：Spark SQL 业务的 Yarn-Cluster 运行模式、On Demand 的 History Server、元数据库代理、Hive UDF/UDAF 函数自动加载等。
- 新功能：数据自动分区、自定义 Python UDF/UDAF 函数、多维分析组合使用等。
- 性能优化：预排序的 Join 实现方式、多维分析的 Union 策略、SortMergeJoin 执行过程中的数据存储优化等。

整个替换升级过程对用户都是透明的，如今，腾讯内部的 Spark SQL 在性能和稳定性上已经能够完全满足各大业务数据处理的需求。

11.2.2 SQL-Engine 整体架构

实际上，完整的 SQL-Engine 包括多个模块，这里为了突出重点，在图 11.2 中仅展示了与 Spark SQL 相关的部分。HiveServer 是服务器，沿用了早期的命名，负责管理应用的生命周期。在

上层，用户可以以两种方式来提交 SQL 语句，一种是基于可视化开发环境的即时查询，另一种是调用数据流系统周期性运行。这两种方式都会通过 Thrift 协议将 SQL 语句提交到 HiveServer 集群。HiveServer 接收到应用后，将应用以 Yarn-Cluster 方式提交到 Yarn 集群运行，并将计算的结果返回给业务。

Spark SQL 在运行过程中，会连接 MetaStore 集群获取元数据信息，以及连接 UDF 集群获取计算所需的 UDF/UDAF 函数。可以看到，SQL-Engine 的架构与业界/社区的一些通用解决方案不同。首先，SQL 语句会通过 HiveServer 在 Spark SQL 中封装为一个应用（Application），然后以 Yarn-Cluster 的形式提交到集群，这种方式可以支持高并发的业务模式。其次，为了避免模块之间的耦合，元数据信息和 UDF 函数在 SQL-Engine 中都剥离出来进行单独管理，这样元数据信息和函数的更新可以对业务保持透明，便于版本升级。

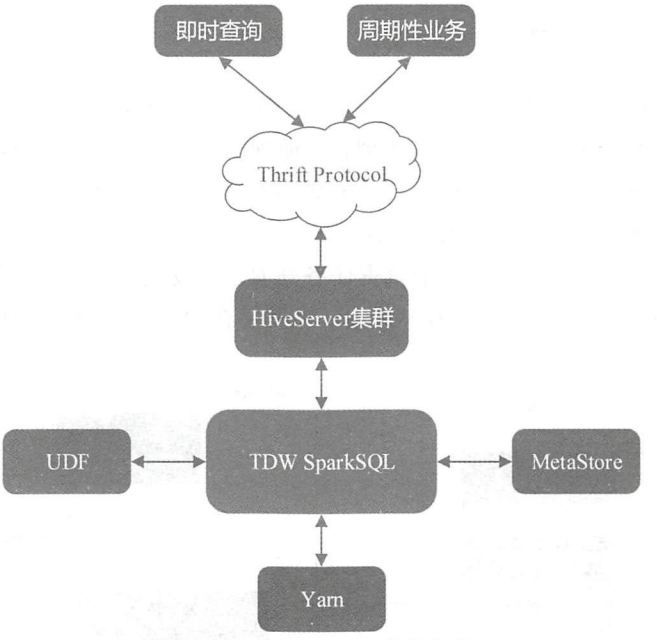


图 11.2 SQL-Engine 整体架构

这种架构可以通过简单的水平扩展增加其业务的处理能力，并且扩展可以对用户业务保持透明。在 HiveServer 内部，通过不同业务间的隔离，可防止有问题的业务影响其他业务。对失败的业务，增加自动重试机制。同时，在提交业务到 HiveServer 时，通过负载均衡算法选择相对空闲的 HiveServer 处理任务。这些机制保证了 SQL 引擎在海量业务压力下的稳定性、容灾能力高性能。

11.3 TDW-Spark SQL 开发与优化

为保证海量数据和业务压力下的高可用和高性能，同时满足不同业务的复杂需求，团队在社区版本的基础上做了大量的改进和优化。本章内容涉及 3 个方面：首先是业务运行的支撑框架，该框架能够确保 Spark SQL 稳定地融入到 SQL-Engine 的整体架构中；其次是新功能的开发，以兼容内部的一些数据特性或需求；最后是性能优化，基于内部执行环境，完成一些针对性的改进。

11.3.1 业务运行支撑框架

1. Yarn-Cluster 提交模式

根据 SQL-Engine 的整体架构可知，HiveServer 负责向 Yarn 提交 SQL（注：腾讯内部使用的是 Gaia，这里使用 Yarn 进行介绍，原理类似，方便理解）。社区的 Spark 支持多种提交方式，如果是默认的 Yarn-Cluster，则 HiveServer 无法有效地和 ApplicationMaster 进行交互，因此从功能上来看，只有 Yarn-Client 模式满足需求。如图 11.3 所示，在 Yarn-Client 模式下，SparkDriver 将运行在 HiveServer 所在的节点上。由于负责任务分发，以及消息、状态的汇总等，Driver 会给 HiveServer 服务器造成非常大的（内存）压力，尤其是在 Driver 获取 mapOutputStatus 对象时。

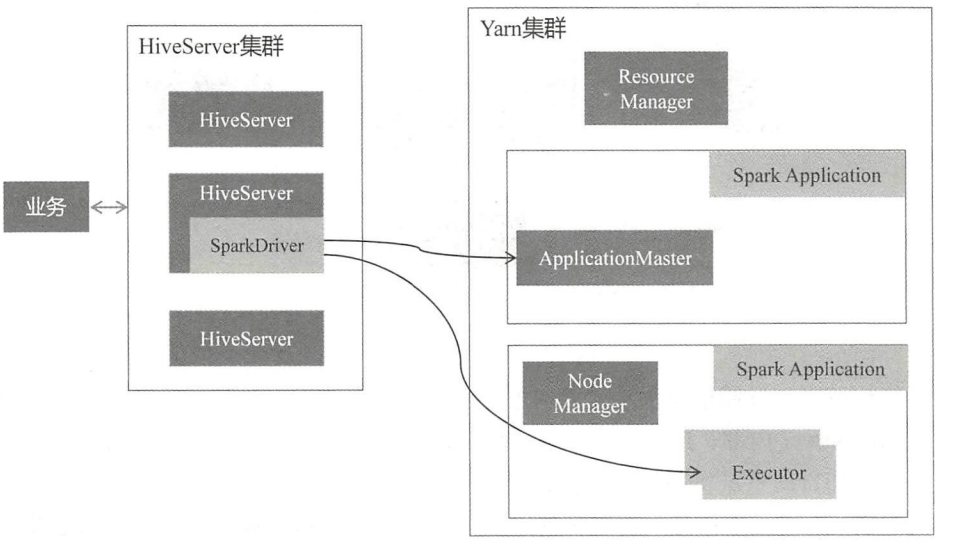


图 11.3　Yarn-Client 模式下的运行架构

针对这些问题，SQL-Engine 采用了以下方案。

（1）抽象出通用的业务无关逻辑，仅暴露 SQL 接口，实现 Spark SQL 对用户透明。

（2）遵循 Spark 提交模块中 Yarn-Cluster 模式的思路，实现 Spark SQL 的 Yarn-Cluster 提交模式，完成 Driver 的分散化，避免单点问题（Driver 即使失败了，Yarn 也会重新分配并启动另一个 Container 来运行 Driver），达到高并发、高可用的目标。

（3）定义一套专用消息接口，实现 Yarn-Cluster 模式下 Client 端与 ApplicationMaster 的消息通信，支持实时动态获取业务 SQL 的执行状态信息。

基于上述解决方案，SQL-Engine 中实现的业务提交模块整体架构如图 11.4 所示，该架构中重要的部分有 3 个。

- 通用程序：通用的基本操作会被封装成应用程序，在 Spark Driver 中只要接收用户提交的 SQL 语句即可。
- SQLApplicationMaster：实现 Spark SQL 系统的 Yarn-Cluster 模式，SQLApplicationMaster 类似 Spark-On-Yarn 提交模式中的 ApplicationMaster 角色，并在 SQLApplicationMaster 中封装 SQL 逻辑为 Yarn 中的应用。
- SQLMessage：实现 Spark SQL 的客户端（Client），定义 SQLApplicationMaster 与 Client 之间的消息接口 SQLMessage，并在 Client 端提供实现的各种命令（如提交 SQL 语句的 submitSql 等）与消息获取方法（如 getState 等）。

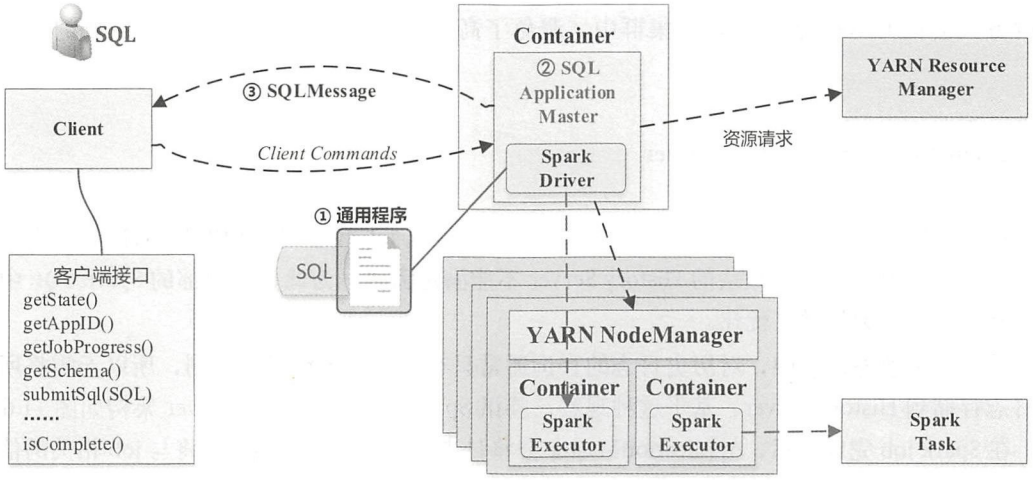


图 11.4 SQL-Engine 中实现的业务提交模块整体架构

SQLApplicationMaster 包含 ApplicationMaster 的全部功能（建立基础的 RPC 上下文环境、保持与 Driver 之间的联系、通过 Event 机制驱动 AMEndpoint、创建 reportThread 与 ResourceManager

通信等），并在此基础上实现了两个重要的逻辑：与客户端保持通信，接收提交的用户命令和查询等；提交封装的通用程序给 Driver 端，将任务执行的状态信息返回给客户端。总的来讲，SQLApplicationMaster 类是整个技术方案的中枢。

针对 Spark SQL 业务中 SQLApplicationMaster 与客户端状态信息传递的需求，单独定义了一套状态消息标准。如图 11.5 所示，状态消息定义为 SQLMessage。该接口的具体实现包括 Driver 注册（RegistedDriver）、用户 SQL 提交（SQLSubmitted）和 Schema 信息获取（CurrentSchema）等 27 种，涵盖了任务运行过程中的各个方面。基于 SQLMessage，HiveServer 通过 AKKA 与 SQLApplicationMaster 之间通信来管理 SQL 执行的生命周期。

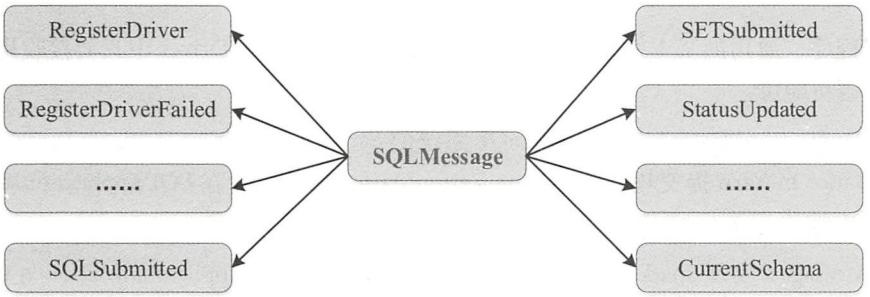


图 11.5　状态消息定义

基于上述方式，Spark 的 Driver 端会运行在 SQLApplicationMaster 所在的 Yarn Container 上。这种方式会把 Driver 分散在 Yarn 集群中，避免了高并发时 Driver 给 HiveServer 造成的巨大压力。

2. On-Demand 的 History Server

由于社区的 Spark History Server 不能水平扩展，所以 Job 较多时查询性能不佳[51]。在腾讯海量业务的压力下，社区默认的 History Server 不能满足需求。为此，在内部的 Spark SQL 中对 History Server 的架构做了优化。

在实际的业务环境中，对历史日志的查询通常只有在业务出错时会用到，所以无需将所有的日志存储到 History Server。基于这种观察，腾讯 Spark SQL 的 History Server 架构如图 11.6 所示。在 Spark Job 完成之后，ApplicationMaster 会将日志文件保存在本地，并将与 job 相关的信息保存到数据库。

当有业务需要获取日志时，History Server 首先从数据库中获得 Job 信息，得到存储日志文件的地址，然后将日志文件复制到 History Server 进行展示。实践结果表明，这种 On-Demand 方式可以满足海量业务下的日志查询需求。

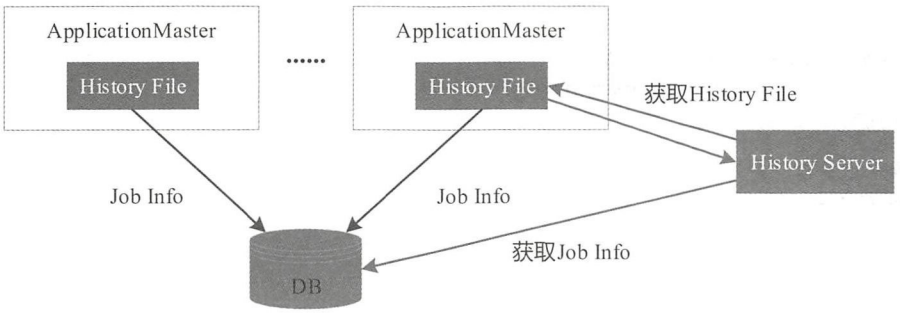


图 11.6 History Server 架构

3. 元数据库代理

为了防止 Spark SQL 大量的元数据请求对元数据库造成很大的冲击，腾讯 Spark SQL 通过代理机器访问元数据库，如图 11.7 所示。Spark SQL 通过 Thrift 协议将请求发送给代理，然后代理将请求发送给元数据库，并将结果返回。这种代理方式可以在访问高峰时起到对元数据库的保护作用。

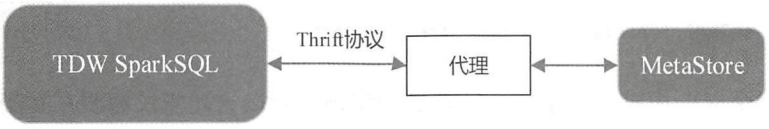


图 11.7 元数据库代理

元数据服务在 Spark SQL 执行过程中起着很重要的作用，在实际生产环境中会有波峰的影响。目前，SQL-Engine 中的元数据已经逐步由单机版本的 PostgreSQL 迁移到内部自研的分布式版本 TBase 数据库。

4. Hive UDF/UDAF 函数自动加载

前面已经介绍过，SQL-Engine 的业务最早都运行在内部版本的 Hive 中，因此业务迁移到 Spark SQL 的过程中需要考虑两个系统在执行行为上的不同。UDF 函数就是其中之一，与社区的实现相比，无论是数量还是功能，内部许多 UDF/UDAF 函数都存在着差异。为了做到函数的实时更新对业务保持透明，SQL-Engine 将 UDF/UDAF 函数从 Hive 从解耦出来，单独维护。

这样，在提交 SQL 任务时，Spark 系统首先自动加载 UDF/UDAF 函数。在解析 SQL 时，会先查找加载的 Hive UDF/UDAF 函数，查找失败时再去查找 Spark SQL 中的内置函数。加载逻辑的代码片段如下。

```
try {
  // 获得Hive UDF/UDAF函数的管理对象
  val pefrh = getPreExecFunctionRegisterHooks
  pefrh.run()
  }
} catch {
  case e: Exception => {
  }
}
```

11.3.2 新功能开发案例

1. 字符串拼接

通常情况下，字符串拼接都是通过 concat 函数来完成的，concat(“a”, “b”) 会得到 “ab” 字符串。在 TDW 中，需要支持用户使用 “||” 完成相同的功能，例如要给学生名字加上前缀，SQL 语句为：

```
select ‘‘prefix’’ || name from student
```

因为在 Spark SQL 中已经有了 concat 函数的实现，所以该功能仅需要在文法层面和逻辑计划层面实现。具体来讲，可分为两步。首先，在文法文件 SqlBase.g4 中添加相应的文法，这里称为 tdwConcat，代码如下。

```
primaryExpression
    : CASE value=expression whenClause+ (ELSE elseExpression=expression)? END    #simpleCase
    ......
    | left=primaryExpression '||' right=primaryExpression    #tdwConcat
    ;
```

这条文法添加在 primaryExpression 的定义中，左表达式和右表达式仍然是 primaryExpression，注意 tdwConcat 与其他标签的先后顺序。其次，就是修改 AstBuilder 类生成对应的逻辑算子节点，需要加入针对 tdwConcat 的逻辑，即重载实现 visitTdwConcat 方法，代码如下，直接得到 UnresolvedFunction，将其对应到 concat 函数。

```
override def visitTdwConcat(ctx: TdwConcatContext): Expression = withOrigin(ctx) {
  val arguments = Seq(ctx.left, ctx.right).map(expression)
  UnresolvedFunction(”concat”, arguments, false)
}
```

2. 灵活的多维分析组合

开源社区的 Spark SQL 只支持多维分析函数 grouping sets、cube、rollup 的单独使用，不支持组合使用，且只能通过关键字 with 选取固定的列。在 TDW 平台涉及的业务中，对多维分析的使用比较灵活，无法直接迁移到 Spark SQL 中，例如下面的 SQL 语句就会直接抛出语法错误。

```
select expr1, expr2, expr3, expr4, expr5, sum(expr6)
from myTable
group by expr1, rollup(expr2, expr3), cube(expr4, expr5)
```

针对该需求，内部的 Spark SQL 系统中加入了对多维分析函数组合使用的支持，大大增加了多维分析的灵活度，满足不同 OALP 业务的需求。

在具体实现上，为了支持多维分析的灵活使用，需要在 SQL 语法中支持。本书第 4 章中已经介绍过 Spark SQL 语法编译器。增加语法支持需要修改 SqlBase.g4 文件中与聚合相关的文法，代码如下。

```
aggregation
    : GROUP BY groupingExpressions+=groupByExpression (',' groupingExpressions+=groupByExpression)*
    ;

groupByExpression
    : cubeExpression
    | rollupExpression
    | groupingsetExpression
    | expression
    ;

cubeExpression
    : CUBE '(' expression (',' expression)* ')'
    ;

rollupExpression
    : ROLLUP '(' expression (',' expression)* ')'
    ;

groupingsetExpression
    : GROUPING SETS '(' expression (',' expression)* ')'
    ;
```

修改后的语法会被翻译为抽象的语法树，语法树再被解析为逻辑计划。所以，为了使多维分析的语法生效，还需要修改执行逻辑。从语法树到逻辑计划（LogicalPlan）是在 AstBuilder 中完成的。在 AstBuilder 中，聚合操作会被翻译成逻辑计划 GroupingSets。GroupingSets 的定义如下。

```
case class GroupingSets(
    bitmasks: Seq[Int],
    groupByExprs: Seq[Expression],
    child: LogicalPlan,
    aggregations: Seq[NamedExpression]) extends UnaryNode {

  override def output: Seq[Attribute] = aggregations.map(_.toAttribute)

  override lazy val resolved: Boolean = false
}
```

GroupingSets 在第 7 章中简单介绍过，其中 groupByExprs 表示聚合的字段，而 bitmasks 用来表示聚合字段的组合。在修改多维分析的执行逻辑时，只需修改这两个参数。使用本小节开头的 SQL 例子，修改后的 groupByExprs 为 [expr1, expr2, expr3, expr4, expr5]，bitmasks 为 (0, 1, 2, 3, 4, 5, 6, 7, 12, 13, 14, 15)。bitmasks 中二进制表示的 0 代表对应字段参与聚合，1 代表对应字段不参与聚合。例如，6 的二进制为“00110”，表示参与聚合的字段为 expr1、expr2、expr5，其他字段为 null。

在修改了 SqlBase.g4 和 AstBuilder 之后，多维分析的组合就可以使用了。这个功能只涉及对语法的支持，不会涉及后续执行层面的改动。上述实现的说明基于 Spark 2.1 版本，在后续版本中，读者可以根据该思路自行实现。

3. 自定义 Python UDF/UDAF 函数

虽然 SQL-Engine 提供了大量的 UDF/UDAF 函数以支持各大业务的数据处理，但是在实际运营中仍然会发现已有的函数无法满足所有的逻辑需求。在这种情况下，最好的方式是支持用户自定义函数，并且能够很方便地被他人复用。

目前，Spark SQL 中主要支持两种形式的 UDF/UDAF 函数自定义，一种是在 Spark 应用程序中定义的一次性函数，另一种是引用 Hive 的 UDF/UDAF 函数。前一种形式定义的函数在目前 SQL-Engine 的架构体系下无法直接使用（用户不需要写程序），而后一种用户自定义函数的门槛比较高，很难推广使用。

实际上，TDW 平台的用户中具有 Python 背景的数据分析人员占很大一部分比例。因此，为了降低用户使用门槛，内部 Spark SQL 中实现了 Python UDF 函数接口和 UDAF 函数接口，用户可以继承接口实现自定义的函数功能。同时，用户实现的函数可以在完成注册后为他人复用。

UDF 函数对数据进行逐行处理，输入一行数据，就会输出一行处理后的数据，输入与输出是一对一的关系。实现 UDF 函数只需实现接口的 eval 函数，代码如下。函数的输入为一行数据，输出为处理后的数据。案例中的函数 StringLength 用来计算字段长度。

```
class UDF(object):
    def eval(selft, data):
    '''
    计算结果，然后返回结果
    '''

# 示例：计算String型字段的长度
class StringLength(UDF):
    def eval(self, data):
        return len(data)
```

UDAF 函数是聚合数据，输入多行数据，输出一行处理后的数据，输入与输出是多对一的关系。实现 UDAF 函数需要实现 4 个接口，代码如下。接口 initialize 是初始化 buffer（用来保存结果）。接口 update 用于处理一行数据，并将处理结果保存在 buffer 中。接口 merge 是将 buffer 中的结果进行部分聚合。接口 eval 返回最终结果。案例函数 Average 用来计算 String 型字段的长度。

```
class UDAF(object)::
    def initialize(self):
    '''
    创建聚合buffer，给定初值，返回buffer
    '''
    def update(self, buffer, data):
    '''
    处理每行数据，结果保存在buffer中，返回buffer
    '''
        return buffer

    def merge(self, buffer1, buffer2):
    '''
    局部聚合部分结果，返回buffer
    '''
        return buffer

    def eval(self, buffer):
    '''
    计算最终结果，返回结果
    '''

# 示例：计算平均值
  class Average(UDAF):

      def initialize(self):
          '''
          创建聚合buffer，buffer[0]保存总和，buffer[1]保存数据的数目
          '''
          buffer = [0.0, 0]
```

```
        return buffer

    def update(self, buffer, data):
        '''
        处理每行数据
        '''
        buffer[0] += data[0]
        buffer[1] += 1
        return buffer

    def merge(self, buffer1, buffer2):
        '''
        聚合部分结果
        '''
        buffer1[0] = buffer1[0] + buffer2[0]
        buffer1[1] = buffer1[1] + buffer2[1]
        return buffer1

    def eval(self, buffer):
        '''
        计算并求均值
        '''
        if(buffer[1] != 0):
            return buffer[0] / buffer[1]
        else:
            return 0.0
```

Python UDF/UDAF 函数需要在 PySpark 的环境下运行。PySpark 是在 Spark 外围包装一层 Python API，如图 11.8 所示。在 Driver 上，Python 进程通过 Py4j 实现在 Python 中调用 Java 的方法；在 Executor 上，为每个 task 启动一个 Python 进程，Python 进程通过 Socket 与 task 进行数据通信。

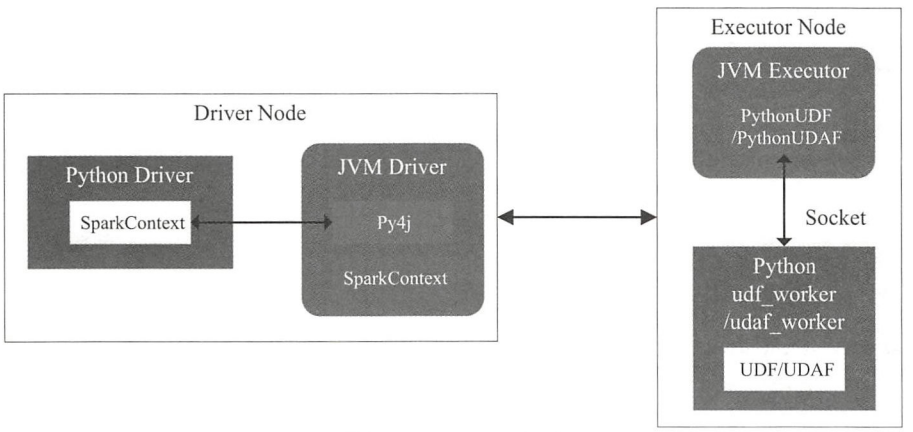


图 11.8　PySpark 架构

具体到 Python UDF/UDAF 函数，task 将需要计算的数据发送给 Python 进程中的 UDF/UDAF 函数进行计算，计算完成后将计算结果返回给 task。为此，内部的 Spark SQL 中实现了 PythonUDF 和 PythonUDAF 两个对象，负责 task 中数据的发送和接收。在 Python 模块里，实现了 udf_worker 和 udaf_worker，用于在 Python 进程中负责数据的接收，并调用相关函数进行计算，然后将结果返回给 task。

4. 自动分区

Spark 任务中如果存在多个 Stage，那么就必然涉及数据的 Shuffle 操作。此时，系统将面临一个如何设置 Shuffle 后的数据分区数目的问题。目前，Spark 系统默认采用统一固定的数据分区数目（200），并且需要手动设置（通过参数 spark.sql.shuffle.partitions 设置）。因此，每次运行 Spark SQL 任务时，用户都需要预估 SQL 要处理的数据量，然后根据 Executor 的内存大小，手动设置数据分区数目。设置不当可能会造成内存 OOM 或资源浪费。

当每个 Stage 都采用统一的分区数目时，为了使分区的数据量适应 task 的内存大小，分区数目需要按照 Stage 中最大的数据量进行设置，这样对于处理数据量小的 Stage，内存资源是浪费的。如图 11.9 所示，Stage 1 处理的数据量为 30GB，Stage 2 处理的数据量为 200GB（假设 Stage 1 的数据没有膨胀），数据分区数目需要按照 Stage 2 处理的数据量进行设置。如果每个 task 能够处理的数据量为 1GB，那么数据分区的数目需要设置为 200。当分区数目为 200 时，Stage 1 中的每个 task 处理的数据量仅为 30/200=0.15GB，造成 task 内存资源的浪费。

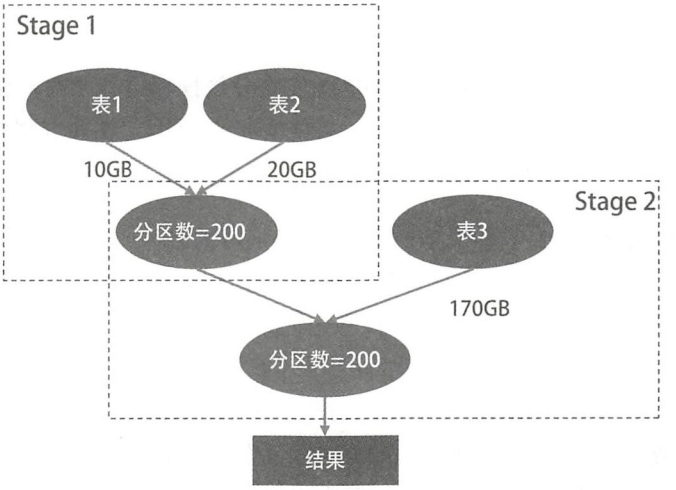


图 11.9 数据分区数目的统一设置

内部 Spark SQL 在社区的基础上对分区设置进行了改进，如图 11.10 所示。在物理算子树生

成的 RDD（图 11.10 中的 RDD5）被提交之前，由负责提交的 SparkContext 对象来触发 RDD 中定义的重分区操作（resetPartition）。通过递归调用父 RDD 的 resetPartition 方法触发所有 RDD 的重分区操作。resetPartition 方法会根据 RDD 输入的数据量（getInputSize）进行分区数的重新计算。每个 RDD 输入的数据量由父 RDD 决定，例如图 11.10 中 RDD3 的数据量从 RDD1 的数据量得到，RDD5 的数据量由 RDD3 和 RDD4 的总数据量得到。

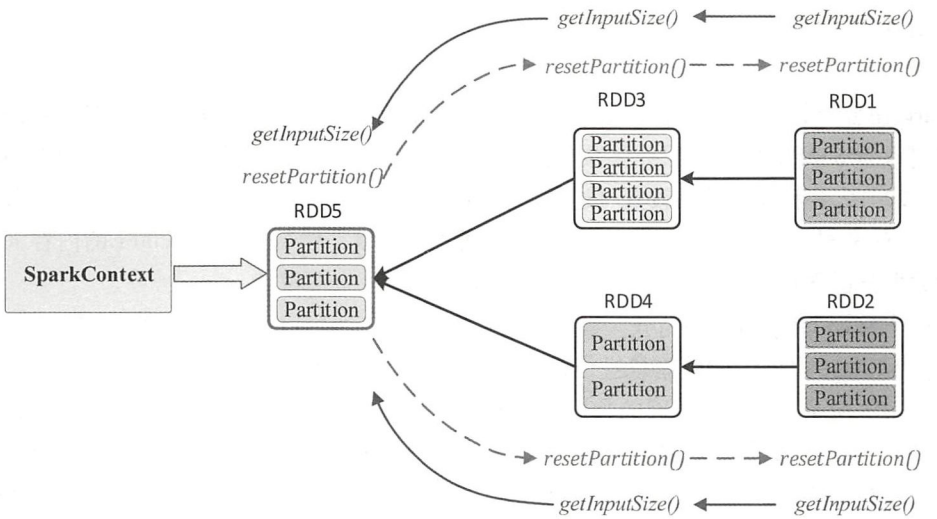


图 **11.10**　自动分区框架

在具体实现中，每个 RDD 实现了 resetPartition 和 getInputSize 两个操作，分别用来进行重分区和获取 RDD 的输入数据量。当一个 RDD 触发重分区操作时，会对其依赖的父 RDD 对象递归地执行重分区操作，一直追溯到负责输入数据读取的 RDD。在此过程中，根据 RDD 之间的物理算子的不同，分为以下两种情况。

- 物理算子不涉及分区操作，即当前的 RDD 与其依赖的 RDD 之间处于分区相同的状态。在这种情况下，假设将当前的 RDD 重分区数目设置为 k，那么其所依赖的所有父 RDD 都执行数目为 k 的重分区操作。
- 物理算子涉及分区操作，即当前的 RDD 与其依赖的 RDD 之间分区并不统一。这种情况的 RDD 类型为 ShuffledRowRDD，物理算子称为 Exchange 节点。如图 11.11 所示，在 Exchange 节点中会生成一个中间 RDD（Internal RDD），中间 RDD 是 ShuffledRowRDD 的父 RDD。针对这种情况，需要在 ShuffledRowRDD 中重新设置 Exchange 中的中间 RDD 的分区对象（Partitioner），使得该分区对象中设置的分区数目与 ShuffledRowRDD 中的分区数目统一。

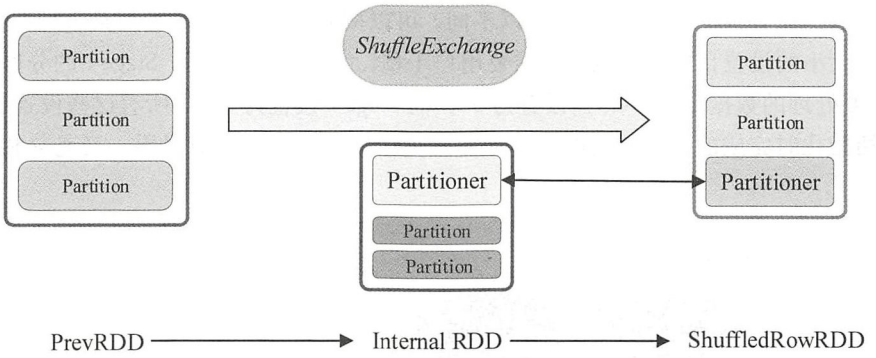


图 11.11 Exchange 分区设置

其中，resetPartition 和 getInputSize 的实现逻辑比较简单，代码如下。在 resetPartition 中，DATASIZE_PER_PARTITION 决定每个分区的数据量，MIN_PARTITIONS 和 MAX_PARTITIONS 分别是分区数目的上限和下限。在某些特定的物理算子节点（例如聚合操作中的 Distinct）上，数据会发生膨胀，在根据输入的数据量计算分区数目后，需要（通过系数 partAdjustment）对分区数目进行调整。getInputSize 通过递归调用，最终由输入型 RDD（例如 HadoopRDD）计算 SQL 最初读入的数据量，HadoopRDD 通过 HDFS 提供的 API 完成计算。

```
def resetPartition() = {
    // 调用父RDD的resetPartition函数
    prev.resetPartition()
    // 计算分区数
    var num = 1 + (getInputSize / DATASIZE_PER_PARTITION ).toInt
    // 对于会发生数据膨胀的，调整分区数目
    num = num * partAdjustment
    if(num > MAX_PARTITIONS)
      num = MAX_PARTITIONS
    else if(num < MIN_PARTITIONS)
      num = MIN_PARTITIONS
    // 设置分区数目
    partitioner.get.resetPartitions(num)
}

def getInputSize() = {
    var inputSize = 0
    // 从依赖的RDD获取数据量大小
    for (rdd <- denpendedRDDs) {
      inputSize += rdd.getInputSize
    }
    inputSize
}
```

改进之后，每个 Stage 的分区数目可以不同，可以按照 Stage 处理的数据量进行个性化设置。所有分区的大小的设置自动化完成，不需要用户干预。如图 11.12 所示，Stage 1 的分区数根据这个 Stage 所要处理的数据量（30GB）设置为 30，而 Stage 2 设置为 200。所有这些设置是在 Spark SQL 任务初始化时完成的。这样的改进既让用户免于烦琐的任务参数设置，又充分利用了内存资源。

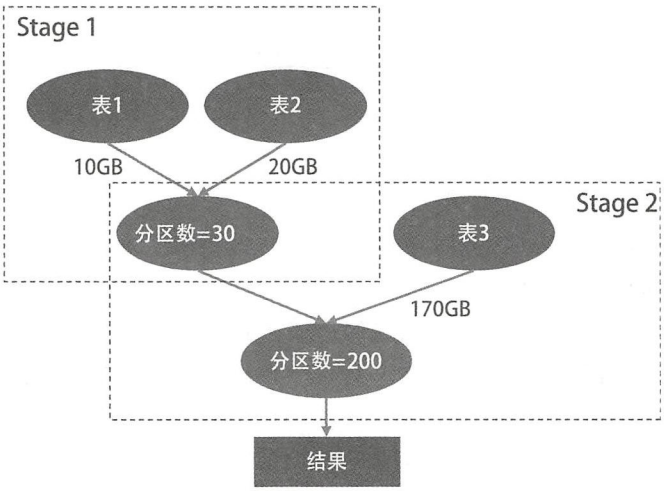


图 11.12　数据分区数目的按需设置

作为服务于不同业务的基础平台，自动分区是必不可少的功能。目前的方案能够基本满足用户需求，但实现方式还较为单一，后续版本会逐步优化重构。实际上，针对该需求，业界也有着各种方案，有兴趣的读者可自行调研。

11.3.3 性能优化开发案例

社区 Spark SQL 的性能总体上能够满足大部分业务的需求，但是随着系统的逐渐迁移，在腾讯海量数据和复杂业务的压力下，Spark SQL 中的一些性能瓶颈开始凸显。这些瓶颈有的是 Spark SQL 中本身就存在的，还有一些是在内部特殊的环境中出现的。针对这些问题，SQL-Engine 团队进行了针对性的优化，涉及从简单的表达式到复杂的执行方式改进等多个层面，下面介绍其中两个开发案例。

1. 预排序的 Join

笔者在第 8.5.4小节中介绍了 Spark SQL 中最常用的 Join 实现机制 SortMergeJoinExec。如图 11.13 所示，在这种机制下，在 Shuffle 之前，Map 阶段会按照 key 的哈希值对数据进行重分

区，相同 key 的数据被分配在同一个分区中，不同 Mapper 中相同分区的数据会被 Shuffle 到同一个 Reducer 中。Reducer 会对来自不同 Mapper 的数据按照 key 进行排序，然后对排序后的数据进行 Join。

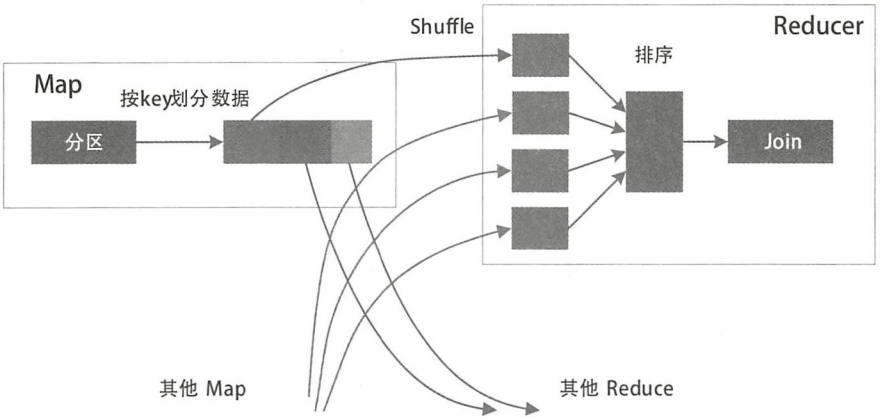


图 11.13 优化前 Join 的实现

这种机制的不足之处是，当 Reducer 数量比较少时，会造成 Reducer 处理的数据量比较大，尤其当出现数据倾斜的情况时（某个/些 Reducer 的数据量远大于其他的 Reducer），排序的性能会受到很大的影响，拖慢 Join 的速度。

内部 Spark SQL 针对 SortMergeJoinExec 的不足之处对其进行了优化，如图 11.14 所示，优化的思路借鉴了 MapReduce，把数据排序提前到了 Map 阶段。Map 阶段会按照 key 的哈希值对数据进行重分区并按 key 排序。Reducer 只需对来自不同 Mapper 的数据进行归并排序。这种机制相当于把 Reducer 排序的任务分流给 Mapper。而由于 Mapper 的数据量往往是比较均匀的，所以排序的性能会优于 Reducer。根据实际观察，在腾讯的某些业务中，这种机制会带来几倍的性能提升。

Map 阶段的数据排序及聚合由函数 joinShuffleWrite 完成。函数的输入为 RDD 的分区数据，然后将每条数据插入到排序器中，直到全部数据都插入完成，最后将所有的排序文件合并。主要逻辑参见下面的伪代码。

```
def joinShuffleWrite(Iterator<Product2<K, V>> records) {
 while (records.hasNext())
  sorter.insertRecord(records.next())
 end while
 mergeSpills()
}
```

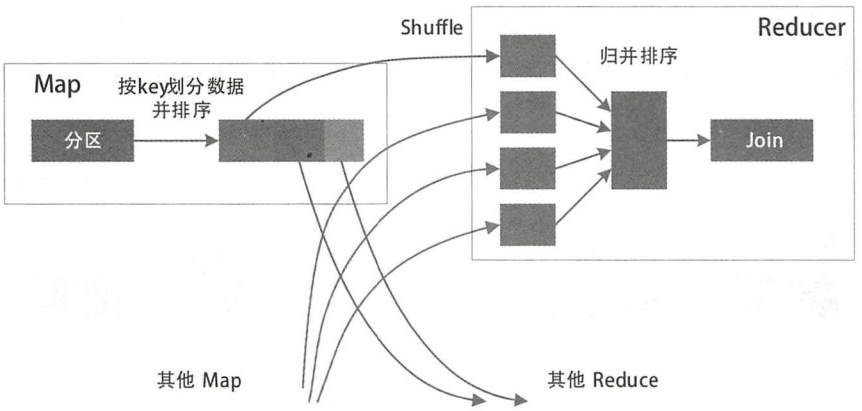


图 11.14　优化后 Join 的实现

其中，insertRecord 方法是将一条数据记录插入到排序器 sorter 中，而 mergeSpills 方法是将排序的文件合并到一个文件中。insertRecord 的逻辑参见下面的伪代码。

```
def insertRecord(Object record) {
 if (memoryBuffer.size() >= threshold)
  sortAndSpill(memoryBuffer)
 end if
 add record to memoryBuffer
}
```

代码中的 memoryBuffer 使用内存存储 record，当达到一定数量时，sortAndSpill 方法将 memoryBuffer 数据排序并存储到磁盘上，同时清空 memoryBuffer 数据结构。

mergeSpills 方法将所有 insertRecord 中的 spilled 文件合并。其逻辑参见下面的伪代码。每次从 spilled 的文件中取出一个属于当前 partition 的最小值并写到文件中，如果没有属于当前 partition 的数据，则换到下一个 partition，直到所有数据都被取出。

```
def mergeSpills() {
 while (currentPartitionId != null)
  get smallest record from spilled files belong to currentPartitionId
  if (record != null)
    write record to output file
  else
    if (has next partition)
      currentPartitionId ← next partition
    else
      currentPartitionId ← null
   end if
  end if
 end while
}
```

Reduce 阶段将不同 Mapper 的数据合并，并以 Iterator 的形式输入到 Join 算子中。这样调用 Interator.getNext() 就可以得到下一个数据值，其逻辑跟 Map 阶段的 mergeSpills 方法类似，代码如下。

```
def getIterator().getNext()
 while (True)
  get smallest record from map files belong to currentPartitionId
  if (record != null)
    return record
  else
    if (has next partition)
      currentPartitionId ← next partition
    else
      currentPartitionId ← null
     return null
   end if
  end if
 end while
}
```

2. 多维分析的 Union 方式

社区 Spark SQL 对多维分析采用 Expand 方式，一次读入数据，读入的每条数据会生成多条（2^n，n 为维度大小）。也就是说，Expand 方式会将数据放大 2^n 倍，然后将放大后的数据进行聚合。例如，下面的这条 SQL 语句。

```
select A, B, sum(C) from myTable group by A, B with cube
```

在 Spark SQL 默认的 Expand 执行方式中，数据将会被放大 4 倍，对应逻辑如图 11.15 所示。每条数据会按照 (A, B) 两列的组合，添加一个 Grouping ID，以区分所属的分组。最终执行等价于在 4 倍数据上的一次聚合操作。

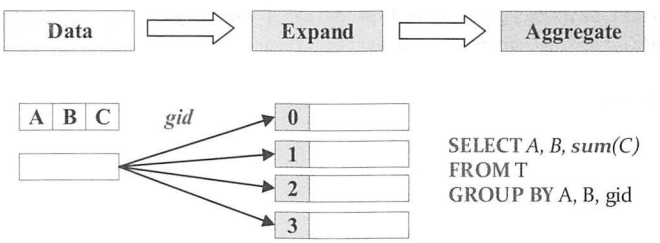


图 11.15 Expand 方式执行的 Cube

然而，当维度比较大或初始数据量比较大时，在这种数据指数性膨胀的执行方式下，数据 Shuffle 的性能往往会非常差，某些资源有限的情形下还会出现 OOM 的问题。因此，团队在 Spark SQL 中增加了多维分析的 Union 实现方式。Union 方式不是一次读入数据，而是读入 2^n 次数据，分别计算，然后将计算的结果合并。对于案例 SQL，Union 方式的思路如图 11.16 所示。可以看到，Union 方式会读取 4 次数据表，然后将 4 次的求和结果合并到一起。

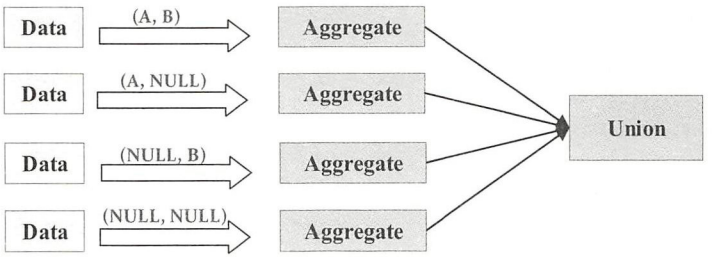


图 **11.16**　Union 方式执行的 Cube

上述逻辑可以实现为一条 SplitAggregateWithExpand 优化规则并加入到 Optimizer 中，当碰到 Expand 与 Aggregate 的组合时，能够将 Expand 拆分成多个 Expand。如以下代码所示，用户可以配置是否开启，以及每个 Expand 中的数据条数。

```
object SplitAggregateWithExpand extends Rule[LogicalPlan] {
  private def splitExpand(expand: Expand, num: Int): Seq[Expand] = {
    val groupedProjections = expand.projections.grouped(num).toList
    val expands: Seq[Expand] = groupedProjections.map { projectionSeq =>
      Expand(projectionSeq, expand.output, expand.child)
    }
    expands
  }
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case a @ Aggregate(_, _, e @ Expand(projections, _, _)) =>
      if (SQLConf.get.groupingWithUnion
        && projections.length > SQLConf.get.groupingExpandProjections) {
        val num = SQLConf.get.groupingExpandProjections
        val subExpands = splitExpand(e, num)
        val aggregates: Seq[Aggregate] = subExpands.map { expand =>
          Aggregate(a.groupingExpressions, a.aggregateExpressions, expand)
        }
        Union(aggregates)
      } else {
        a
      }
  }
}
```

Union 方式和 Expand 方式各有优缺点，Expand 方式读取数据的次数只有一次，但数据会膨胀 2^n 倍，而 Union 方式会读取数据 2^n 次。总体来说，Expand 方式适合维度小的多维分析，Union 方式适合维度大的多维分析。本质上，Union 方式是一种“细水长流”的策略，与常见的 IO 复用等优化思想相反。根据实际观察，在腾讯某些高纬多维分析业务中，Union 方式的执行速度比 Expand 方式快 6~8 倍。目前，这两种方式还需要手动配置，后续发展中会加入这两种方式的智能选取功能，对用户保持透明。

11.4 业务实践经验与教训

本节分享腾讯公司在 Spark SQL 集群管理和业务运营方面的一些经验。虽然这些经验来自于腾讯内部业务，但在一定程度上具有通用性，广大开发人员可以借鉴。

11.4.1 Spark SQL 集群管理的经验

1. Spark SQL 透明更新

在大量业务的情况下，Spark SQL 的版本更新要尽可能减少对用户业务的影响。基于图 11.2 展示的架构，内部 Spark SQL 以 Jar 包的形式存放在 HDFS 上（路径由 spark.yarn.jars 指定）。在版本更新升级时，只需替换 HDFS 上的 Jar 包，无需停止服务，并且能够做到对用户业务完全透明，没有影响。

2. 开启 Executor 动态调整

Spark 在 2.0 版本之后，Executor 动态调整机制已经很稳定了。开启 Executor 动态调整（spark.dynamicAllocation.enabled=true），可以让用户免于烦琐的 Executor 数目的预估和设置，增加业务运行的稳定性。注意，在设置了 Executor 动态调整之后，还需要开启 Shuffle Service（spark.shuffle.service.enabled=true），用来保留被移除的 Executor 产生的 Shuffle 文件。而保留的 Shuffle 文件可用于 Stage 失败的恢复。

3. Executor 动态调整的权衡

Executor 动态调整有 3 个重要的参数：spark.dynamicAllocation.executorIdleTimeout、spark.dynamicAllocation.initialExecutors 和 spark.dynamicAllocation.maxExecutors。

spark.dynamicAllocation.initialExecutors 决定了 Executor 初始数目，当这个值比较小时，任

务需要等待向 Yarn 申请资源，造成任务运行有比较长的爬坡阶段；当这个值比较大时，对于不需要这么多 Executor 的任务来说，会造成资源的浪费。这个值的选取可以根据历史任务的 Executor 数目的统计，按照二八原则来设置，例如 80% 的历史业务的 Executor 数目都不大于参数值。同时，也要考虑集群的资源紧张度，当资源比较紧张时，这个值需要设置得小一点；当资源比较富裕时，这个值可以设置得大一点。

spark.dynamicAllocation.maxExecutors 决定了业务最大可以拥有的 Executor 数目。社区的默认值是无穷大。为了防止大业务独占大部分资源，造成小任务没有资源的情况，需要将这个值设置为一个合理值。这个值的选取需要根据大业务所需的资源情况来设置，防止值过小导致任务执行时间超出业务要求。

spark.dynamicAllocation.executorIdleTimeout 决定了 Executor 空闲多长时间后会被动态删除。当这个值比较小时，集群资源会比较充分地共享，但会影响业务的执行时间（在 Executor 被删除后，可能需要重新申请新的 Executor 来执行 task）；当这个值比较大时，不利于资源的共享，若一些比较大的任务占用资源，迟迟不释放，就会造成其他任务得不到资源。这个值的选取需要在用户业务的执行时间和等待时间上做一个权衡。需要注意的是，当 spark.dynamicAllocation.maxExecutors 为有限值时，spark.dynamicAllocation.executorIdleTimeout 过小会导致某些任务不能申请新的资源。例如 maxExecutors = 10，而某个业务所需的资源大于或等于 10 个 Executor，业务在申请到 10 个 Executor 之后，申请到 Executor 由于空闲（有可能因为 task 还没来得及分配到其上）而被删除，目前社区 Spark SQL 的逻辑是不会再申请新的 Executor 的，这样就会导致任务执行速度变慢。

4. 小文件合并

社区的 Spark SQL 在将数据插入 Hive 表中时，文件数目与 Reducer 数目相同。当 Reducer 数目比较多时，这种机制会造成小文件过多的情况。当读取由大量小文件构成的 Hive 数据时，文件的打开和关闭使得 Spark SQL 读取效率很差。腾讯内部的 Spark SQL 在写数据到 Hive 表时，会增加一个步骤——将小文件合并。对于不同时间写入 Hive 表的数据，后台会定期进行数据压缩和合并，这样既可以避免文件多造成的读取效率差的问题，又可以提高存储效率。

5. 冷热数据分离

冷热数据分离是业界常用的数据管理方案。把不常被访问的数据存储到性能低的机器中，增加数据块的大小，减少数据备份数目，采用压缩率高的压缩，提高数据存储效率；常被访问的数据存储到性能高的机器中，减少数据块的大小，增加数据备份的数目，采用解压速率快的压缩，提高数据访问效率。

11.4.2 Spark SQL 业务层面调优

即使是各种成熟的关系数据库系统，也无法在内部自动处理所有情况。在 Spark SQL 中，因为本身的实现机制和更复杂的分布式执行环境，当出现某些业务执行效率低甚至失败的场景时，就需要从业务层面进行调整或优化。在 TDW 实际运营中，出现了大量的这类需求，本小节分享几个较典型的案例。

1. 多表 Join 顺序调整

多表 Join 操作在实际业务中非常常见，主要用来整合多张数据表的信息以支持进一步的分析。在多表 Join 的场景中，数据表的顺序对性能影响比较大。

实际上，多表 Join 一直都是数据库中基于代价优化机制（Cost-Based Optimization，简称 CBO）的重点针对对象。考虑到 Spark SQL 中的 CBO 尚未完全成熟，不能对 SQL 中 Join 的顺序做智能的调整，所以在写 SQL 时需要从业务层面加以判断。

这里分享一个 TDW 平台中的实际案例，如图 11.17 所示，用户需要将数据表 A（30 亿条）、数据表 B（120 亿条）和数据表 C（1 亿条）通过 Join 操作连接起来。原有的写法是表 A 和表 B 先进行 Join 操作，然后与表 C 再进行 Join 操作。经过观察，表 A 与表 B 执行 Join 操作之后得到的结果有 25 亿条数据，所以最终是这 25 亿条数据与 1 亿条数据的表 C 进行 Join 操作。

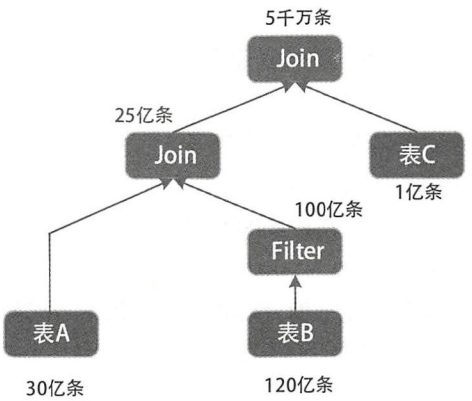


图 11.17 调整顺序前的 Join

经过测试，如果表 A 与表 C 先进行 Join 操作，如图 11.18 所示，Join 的中间结果只有 5 千万条数据，然后与表 B 再进行 Join 操作。这种方式可以大大减少参与 Shuffle 的数据量，提升 Join 的执行速度，性能上得到了大约 40% 的提升。

总的来讲，多个表进行 Join 操作时，总会有一个最佳的执行顺序，从业务层面进行调整需

要对数据的分布情况有大致的了解。目前，TDW 平台在吸收社区优化思想的同时，也在进行适合自身的技术探索。

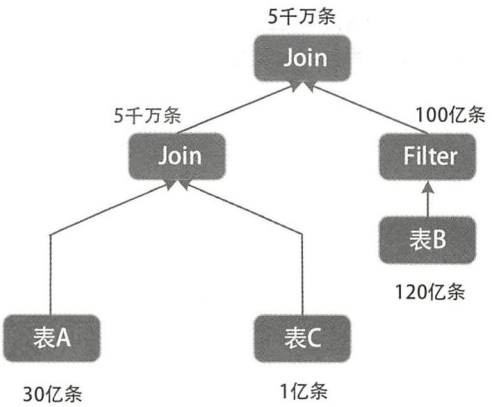


图 11.18　调整顺序后的 Join

2. 数据复用

“复用”是数据处理系统中常用的优化手段，主要包括数据复用和计算复用两个方面。计算复用指的是重用相同的操作逻辑，减少 CPU 的计算代价。数据复用比较好理解，如果读取同一份数据的两个任务之间没有依赖关系，就可以想办法合并任务逻辑，使得只需要读取一次数据，减少 IO 代价。

在 TDW 的业务中，数据复用出现的场景比较多，绝大部分以 Union 操作为主。例如，同一条 SQL 语句模板，用户程序中输入参数，每条 SQL 语句执行后得到一个结果，最后将结果合并起来。此外，即使在单条 SQL 语句中，用户的 SQL 语句也可能用 Union 划分成多个子查询逻辑。

这里举例说明，简单的 SQL 写法如下，用户需要分别选取 key 为 20170802 和 20170803 的数据进行进一步处理（注：这里省略了复杂的处理逻辑，仅用 value 替代）。这条 SQL 语句在 Spark SQL 中执行时，数据表 myTable 会被读取两次，当 myTable 数据量非常大时，对性能有着很大的影响。

```
select  *
from  (
            select value from myTable where key = 20170802
            union all
            select value from myTable where key = 20170803
        )
```

上述 SQL 语句可以在业务层针对性地优化为下面的写法，将筛选数据的逻辑整合在一起。对于进一步的复杂操作，可以用 case when 语句来支持。这样，Spark SQL 只需要读取数据表 myTable 一次，减少了 IO 代价，特别是在存储与计算分离的架构下，能够减少数据传输过程中对网络的占用。

```
select *
from   (
           select value from myTable where key = 20170802 or key = 20170803
       )
```

目前，Spark SQL 在系统层面还缺少这些优化的逻辑，因此需要在业务层面进行更多的调整优化。实际上，复用技术的各种场景已经有了很多研究工作，包括 MapReduce 环境下的 MRShare 和 Stubby 等，都可以借鉴在 Spark SQL 中。

3. Window 函数执行性能优化

一般情况下，当 Spark 集群某个节点的 Task 执行非常缓慢甚至出现 OOM 时，可以适当地增加该节点内存，以提高执行效率。然而，在实际应用中，还存在一类比较特殊的情况。例如，某数据分析业务需要统计每个区域的前若干名用户，SQL 语句中会涉及 row_number 这样的 Window 函数。

在 Spark 执行过程中发现，存在某个分区（partition）的数据量达到 1 亿多条，总共有大约 6.5GB 的数据量。在一定的范围内（注：TDW 设置在 10GB 以内）无论如何增加 Executor 的内存，该分区对应的 Task 最终都会发生内存溢出，导致 SQL 最终失败。

实际上，这个问题和 Shuffle 读写过程中内存的使用方式有关。回顾一下，Shuffle 写的过程中有一个阈值（threshold），超过了这个阈值之后，会将数据文件排序后 spill 到磁盘上形成文件。如图 11.19 所示，假设 threshold 为 4，那么数据相应地写成 4 个文件。而在 Shuffle 读阶段会针对每个数据文件构造一个 reader（参见 UnsafeSorterSpillReader），每个 reader 会申请 1MB 的内存空间作为缓冲区（注：对应代码段“byte[] arr = new byte[1024 × 1024]”）。问题就出在这里，WindowExec 执行计划中数据结构默认设置的阈值是 4096，那么 1 亿条数据会对应产生 24000 多个文件，需要申请 24GB 左右的内存空间，导致了内存溢出的问题。

解决方法比较简单，将参数 spark.sql.windowExec.buffer.spill.threshold 直接调大。这里关于阈值的设置有一个设置范围需要估算一下，假设 Executor 实际 JVM 堆内存大小为 M(GB)，处理的数据条数为 n，大小为 $size$(GB)，那么 $threshold$ 的设置需要大致满足以下两个约束条件：（1）不超过 Executor 内存，即 $size*threshold/n<M$；（2）读取 Shuffle 文件的缓冲区不超过 Executor 内存，即 $n/(1024*threshold)<M$。针对上述例子，如果 Executor 内存设置为 M=4，得到 25000<$threshold$<6100000+，在此范围内选择一个数值即可。

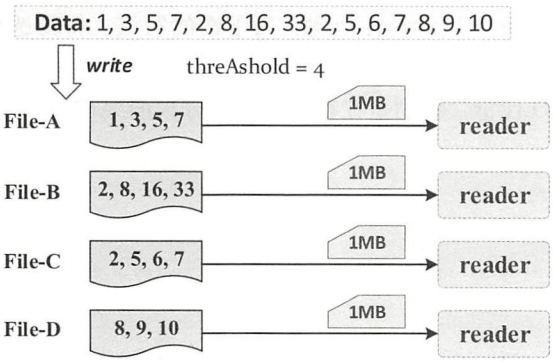


图 11.19　Window 执行 spill 参数调优

从某种程度上看，这种案例可以归纳为一大类。虽然 Spark 系统中有统一的内存管理功能，通过 MemoryManager 实现对一系列 MemoryConsumer 有效的管理。然而，仍然有一些对象属于“黑户”状态，并没有实现 MemoryConsumer 接口。面对这类问题时，需要对 Spark 执行原理有较深入的了解。

4. 数据倾斜处理

在分布式环境中，数据倾斜一直以来都是“痛点”。然而，根据“二八原则”，实际业务中的数据一般都难以均匀分布，例如某个 ID 的用户活动特别频繁、某个时间段系统登录的人数特别多等。

具体到 SQL 语句中，数据倾斜出现的算子主要是 Aggregation 和 Join，其中 Aggregation 中因为有着 Partial 机制，问题并不突出。这里以 Join 为例，当某个或某些 key 的数据量远大于其他时，处理这些 key 的任务运行时间远大于处理其他 key 的任务运行时间，从而拖慢整个 Join 的执行时间。

在 TDW 平台建设的过程中，数据倾斜也是经常碰到的一类问题，在业务层面一般通过以下几种方法来优化或规避。

- 无关数据过滤：经过运维和开发人员的观察，发现实际情况中大约 50% 的数据倾斜都是业务无关的数据导致的，具体分为两种情形：大量的 null 数据没有过滤，参与了 Join 的执行（注：TDW 中根据业务需求没有跳过 null）；存在“脏数据”，不满足原有的数据类型，经过内在的逻辑处理往往会得到 null 等相同结果。这两种情况导致的数据倾斜处理起来比较简单，直接排查后过滤这些无关的数据即可。
- 小表广播：如果参加 Join 操作的两个表是大小表，则可以采用 BroadcastJoin 的方式（参见 8.5.2小节），将小表广播到大表所在的 Executor 上，避免数据倾斜的出现。社区 Spark

从 2.2 版本之后支持在 SQL 中通过添加 Hint 的方式强制采用 BroadcastJoin。例如，下面的 SQL 语句在执行中会将小表 t1 广播到大表 t2 所在的 Executor 上。这种方式处理的情况有限。此外需要注意，对于外连接，基表不能被广播，因此左外连接中左表不可以是小表，右外连接中右表不可以是小表。

```
select /*+ BROADCAST (t1) */ * from t1, t2 where t1.key = t2.key
```

- 倾斜数据分离：例如参加 Join 操作的两个表分别为 t1 和 t2，有数据倾斜的表为 t1。可以将 t1 的数据分为两部分 t11 和 t12，t11 中不包含数据倾斜的数据，t12 中只包含数据倾斜的数据。数据表 t11 和 t12 分别与 t2 进行 Join 操作，然后将结果合并，对应的 SQL 写法如下。一方面，t11 与 t2 的 Join 操作不存在数据倾斜的问题。另一方面，由于数据表 t12 通常不会很大，所以 t12 与 t2 的 Join 操作可以采用第二种方法执行 BroadcastJoin。这样处理之后，数据表 t1 与数据表 t2 的 Join 操作就能够处理数据倾斜的情况了。

```
select * from (
  select * from t11, t2 where t11.key = t2.key
  union all
  select /*+ BROADCAST (t12) */ * from t12, t2 where t12.key = t2.key
)
```

- 数据打散：主要思想在于分散倾斜数据，举一个简单的例子，如图 11.20 所示。假设表 A 和表 B 都有 id、value 字段，需要对表 A 和表 B 按照 id 进行 Join 操作，即“A.id = B.id”。此时，因为 id 都为 a，所有的数据会在一个 task 上进行关联操作，这样就出现了数据倾斜。大数据量的情况下这个 task 将拖慢整个应用的执行效率。数据打散的处理方法就是将大表（A）中的 id 加上后缀（“id_0”-“id_2”），起到“打散”的作用。为了结果正确，小表 B 中的 id 需要将每条数据都“复制”多份。此时再执行 join 操作，将会产生 3 个 task，每个 task 只需要关联一条数据，起到了分散的作用。具体到 SQL 写法，每个数据表的代码如下。

```
表A:
SELECT id, value, concat(id, (rand() * 10000)%3) as new_id
FROM A
表B:
SELECT id, value, concat(id, suffix) as new_id
FROM (
   SELECT id, value, suffix
   FROM B Lateral View explode(array(0,1,2)) tmp as suffix
)
```

经过处理之后，再使用 new_id 作为聚合条件。需要注意的是，这里的 rand 函数的效果不一定非常均匀，后缀数量可以根据实际业务的数据分步来权衡。处理数据倾斜时，有必要先用

count(*) 查看数据的分步情况。此外，可以实现一个 UDF 函数用于专门生成 0 到 n 的数组，便于添加后缀。

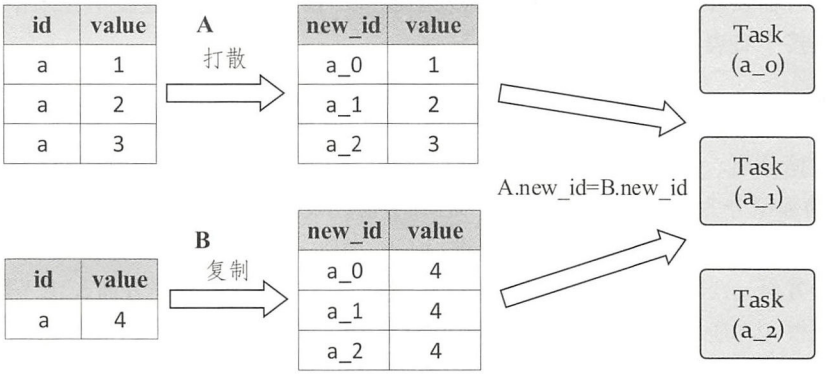


图 11.20　数据倾斜处理

总的来看，业务层面的优化涉及的知识点比较多，存在一定的技术门槛。对于开发人员来说，每个业务层面优化中的思路，都可以进一步地实现在 Spark SQL 系统内部，使得平台处理能力更加强大。

11.4.3 SQL 写法的“陷阱”

除 11.4.2 小节关于业务层面的优化案例外，TDW 平台建设的过程中还积累了大量 SQL 写法“陷阱”方面的案例。这类“陷阱”具有通用性，不仅局限于 Spark SQL 系统。下面介绍几类较典型的例子。

1. null 的特殊性

SQL 标准对 null 的处理规定比较特殊，一些计算逻辑中经常会直接跳过 null。但是在实际情况中，用户经常将 null 当作一个具体的值，因此会出现一些困惑的情况。例如，null 既不参与 IN 表达式的计算，也不参与 NOT IN 表达式的计算，如果数据中存在 null，则这两个表达式得到的结果之和并不等于总的数据结果。

此外，TDW 平台中出现的案例还包括一些函数对 null 的处理。假设数据表 T 中只有一列，列名为 qq，一共有 10 行，数据全部都是 null。那么执行以下两条 SQL 语句，得到的结果并不相同，count(qq) 在执行过程中，会跳过 null。

```
select count(*) from T; //结果为10
select count(qq) from T; //结果为0
```

2. 不确定性 (Non-deterministic) 的 SQL 语句

不确定性语句是指 SQL 中存在每次执行得到的结果不一定相同的表达式或算子（如 rand 函数等）。此外，还有一种情况是由分布式环境下数据 Shuffle 导致的。从用户角度来说，同一条 SQL 语句，似乎每次执行得到的结果都不一样。在实际运营中，容易直接定位的是 rand 等 UDF 函数，后一种情形则比较隐蔽，需要仔细分析其中的细节。这里列举两个 TDW 平台曾经出现的案例。

（1）collect_set 函数 + 数据下标作为判定条件：下面这几条 SQL 语句将数据 value_array 里的数据顺序作为判定条件。而在分布式环境下，数组 value_array 里的数据顺序是不确定的，所以这条 SQL 语句执行的结果也是不确定的。合理的写法是先将 value_array 中的数据排序，然后通过下标获取其中的数据。

```
SELECT id
FROM (
      SELECT id, collect_set(DISTINCT value)  as value_array
      FROM test_table
      GROUP BY id
    )
WHERE value_array[0] = 1
```

（2）Window 函数不确定性问题：下同这几条 SQL 语句中也涉及数据的 Shuffle，Window 函数（row_number）计算之前，需要根据 x 将属于同一个分区的数据 Shuffle 到同一个节点上，但是数据传输有先后顺序，相同 y 的数据到达的顺序不一致，这样即使在 order by 操作之后，整条数据的 row_number 值也是不确定的。

```
SELECT id, row_number() over (partition by x order by y)  as num
FROM test_table
```

总而言之，不确定性除 SQL 语句本身外，还与底层执行环境有关，遇到这种困惑需要具体问题具体分析。目前，TDW 平台正在开发的 Profiling 工具中已经包含了对这种情况的提前检测，用户的 SQL 语句提交之前就会给出一些警告信息。

3. Outer 类型 Join 中的谓词下推

Outer 类型的 Join 操作在实际业务中的应用非常广泛。然而，不同于常规的 Join，Outer 类型 Join 操作的谓词下推的处理比较复杂，用户在写 SQL 语句时非常容易忽略，使得执行结果与自己的本意不符。TDW 平台中曾经出现过多个案例，下面详细介绍谓词下推的几种处理逻辑。

对于 Outer Join，假设返回所有行的基表为 Preserved row table，另外一张表为 Null supplying table，例如 t1 left join t2，则 t1 为 Preserved row table，t2 为 Null supplying table。如果 Join 条件

表达式为“on t1.key = t2.key and t1.key > 1 where t2.key > 2”，则“t1.key > 1”叫作“Join 中条件”，“t2.key > 2”叫作“Join 后条件”。总结起来，Outer Join 语句的谓词下推有 4 种情况，如表 11.1 所示。

表 11.1　Outer Join 语句的谓词下推的 4 种情况

	Preserved row table	Null supplying table
Join 中条件	谓词不下推	谓词下推
Join 后条件	谓词下推	谓词不下推

为了方便分析，构造如下数据，假设表 t1 和表 t2 中的数据相同，都只包含两条数据。下面以数据表 t1 和 t2 为例，说明这 4 种情况。

key	value
1	1
2	2

(1) Preserved row table “Join 后条件”下推

```
select t1.key, t1.value, t2.value
from t1 left join t2 on t1.key = t2.key
where t1.key > 1
```

这种情况下，过滤条件可以下推，SQL 语句等价于：

```
select t1.key, t1.value, t2.value
from (select key, value from t1 where t1.key > 1) t3
left join t2 on t3.key = t2.key
```

SQL 最终执行的结果为：

t1.key	t1.value	t2.value
1	1	1

(2) Preserved row table “Join 中条件”不下推

```
select t1.key, t1.value, t2.value
from t1 left join t2 on t1.key = t2.key and t1.key > 1
```

这种情况下，过滤条件不会下推，SQL 最终执行的结果为：

t1.key	t1.value	t2.value
1	1	null
2	2	2

(3) Null supplying table “Join 中条件” 下推

```
select t1.key, t1.value, t2.value
from t1 left join t2 on t1.key = t2.key and t2.key > 1
```

这种情况下，条件可以下推，SQL 语句等价于：

```
select t1.key, t1.value, t2.value
from t1 left join (select key, value from t2 where t2.key > 1) t3 on t1.key = t3.key
```

SQL 最终执行的结果为：

t1.key	t1.value	t2.value
1	1	null
2	2	2

(4) Null supplying table “Join 后条件” 不下推

```
select t1.key, t1.value, t2.value
from t1 left join t2 on t1.key = t2.key where t2.key > 1
```

这种情况下，条件无法下推，SQL 最终执行的结果为：

t1.key	t1.value	t2.value
2	2	2

在实际运营中，如果用户的 SQL 语句中出现上述“陷阱”，通常会导致结果不在其预期之内。从平台的角度来看，这类问题的根源在于用户对 SQL 标准没有深入理解，需要在相关的文档建设方面做好工作。

11.5 本章小结

本章详细介绍了 Spark SQL 在腾讯公司内部的实践情况。首先，作为背景，介绍了腾讯大数据平台 TDW 的整体组成，并重点回顾了 SQL 引擎的演化历程和架构方式；然后，根据具体的业务需求，重点介绍了基于 Spark SQL 的若干功能开发与性能优化实例；最后，结合具体案例，分享了 Spark SQL 集群管理和业务优化的一些经验。当然，无论是开源社区还是腾讯公司内部，Spark SQL 都在不断地发展中，希望本章的内容能够对读者有所启发，使读者在学习和使用 Spark SQL 的过程中少走弯路。

总结

至此，本书的内容已经全部介绍完毕，下面从头回顾一下整本书的思路与脉络。第 1 章从纵向角度（Spark 到 Spark SQL 的发展）和横向角度（SQL-on-Hadoop 系统）介绍了 Spark SQL 的诞生背景。第 2 章简要介绍了 Spark 架构、编程模型 RDD 和用户编程接口。第 3 章是全书内容的总纲，以一个简单的案例作为研究对象，展示了 Spark SQL 整个执行过程中涉及的几个重要阶段。第 4 章内容集中在领域语言工具 ANTLR 和 Spark SQL 的编译模块。第 5 章重点分析 SQL 逻辑计划阶段的实现，包括逻辑计划的生成和优化。第 6 章分析了 SQL 物理计划阶段的相关实现，包括物理计划的生成、优化、执行准备和最终转换为 RDD 的执行过程。

从本书第 7 章开始，主要内容以专题讨论为主。第 7 章主要讲解聚合查询（Aggregation）的实现，聚合操作是数据分析领域不可或缺的功能，除基本的聚合功能外，本章还包括窗口函数（Window）和多维分析（Grouping Sets/Rollup/Cube）的实现。第 8 章聚焦于经典的 Join 执行，这也是 SQL 中代价比较“昂贵”的操作，本章内容除介绍 Join 相关的逻辑计划与物理计划外，还分析了 Join 各种执行方式的实现，包括 Broadcast Join、Hash Join 和 Sort Merge Join 等。第 9 章侧重于底层，用本书最多的篇幅重点剖析了 Tungsten 计划中的各项优化实现，这也是 Spark SQL 系统中非常关键的部分。第 10 章分析了 Spark SQL 和 Hive 进行交互的实现，包括元数据、数据类型和 Thrift Server 等，对于系统从 Hive 迁移的企业来说，这部分内容必不可少。第 11 章是一些开发实践的经验分享，简单介绍了 Spark SQL 在腾讯 TDW 平台上的应用，并提供了几个典型的开发优化案例。

在大数据时代，各式各样的计算范型、处理框架和平台系统层出不穷，百花齐放，百家争鸣，令人眼花缭乱，即使面对同样的业务场景有时也难以做出选择。然而，万变不离其宗，背后所涉及的分布式技术、编译原理、数据库理论等基础知识都是通用的。作为开发人员，抓住这些系统背后通用的核心技术才是根本。Spark SQL 作为 Spark 技术体系中最为活跃的子项目，发展迅速，其内部机制也越来越复杂，许多技术具有广泛的借鉴意义，既涉及传统数据库和数据仓库系统中的各种基本原理与优化手段，又包含一些新兴的大数据处理技术。当然，Spark SQL 本身也在不断地发展完善中，最好的方式就是跟踪最新的开发动向，并参与到开源社区的贡献之中。总而言之，希望读者读完本书后，能够有所收获。

参考文献

[1] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003: 29–43.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. Proceedings of the 6th Symposium on Operating System Design and Implementation, 2004: 137–150.

[3] F. Chang, J. Dean, S. Ghemawat, et al. Bigtable: A Distributed Storage System for Structured Data. Proceedings of the 7th Symposium on Operating Systems Design and Implementation, 2006: 205–218.

[4] Apache Spark. https://spark.apache.org/.

[5] Apache Hadoop. http://hadoop.apache.org/.

[6] B. Hindman, A. Konwinski, M. Zaharia, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, 2011.

[7] R. S. Xin, J. Rosen, M. Zaharia, et al. Shark: SQL and Rich Analytics at Scale. Proceedings of the ACM SIGMOD International Conference on Management of Data, 2013: 13–24.

[8] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. PVLDB, 2009, 2(2): 1626–1629.

[9] M. Armbrust, R. S. Xin, C. Lian, et al. Spark SQL: Relational Data Processing in Spark. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015: 1383–1394.

[10] V. K. Vavilapalli, A. C. Murthy, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. ACM Symposium on Cloud Computing, 2013: 5.

[11] J. E. Gonzalez, R. S. Xin, A. Dave, et al. GraphX: Graph Processing in a Distributed Dataflow Framework. 11th USENIX Symposium on Operating Systems Design and Implementation, 2014: 599-613.

[12] X. Meng, J. K. Bradley, B. Yavuz, et al. MLlib: Machine Learning in Apache Spark. Journal of Machine Learning Research, 2016, 17: 34.

[13] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Commun. ACM, 1970, 13(6): 377–387.

[14] D. D. Chamberlin and R. F. Boyce. SEQUEL: A Structured English Query Language. Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, 1974, 2: 249–264.

[15] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. PVLDB, 2014, 7(12): 1295–1306.

[16] D. Abadi, S. Babu, F. Ozcan, et al. Tutorial: SQL-on-Hadoop Systems. PVLDB, 2015, 8(12): 2050–2051.

[17] N. Poggi, J. L. Berral, T. Fenech, et al. The state of SQL-on-Hadoop in the Cloud. 2016 IEEE International Conference on Big Data, BigData 2016, 2016: 1432–1443.

[18] Impala. https://impala.apache.org/.

[19] Presto. https://prestodb.io/.

[20] Apache Hbase. https://hbase.apache.org/.

[21] B. Saha, H. Shah, S. Seth, et al. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015: 1357–1369.

[22] C. Olston, B. Reed, U. Srivastava, et al. Pig Latin: a Not-So-Foreign Language for Data Processing. Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, 2008: 1099–1110.

[23] S. Agarwal, B. Mozafari, A. Panda, et al. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. Eighth Eurosys Conference 2013, EuroSys '13, 2013: 29–42.

[24] Apache Hama. https://hama.apache.org/.

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, et al. Pregel: a System for Large-Scale Graph Processing. Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, 2010: 135–146.

[26] Y. Bu, B. Howe, M. Balazinska, et al. Haloop: Efficient Iterative Data Processing on Large Clusters. PVLDB, 2010, 3(1): 285–296.

[27] M. Zaharia, M. Chowdhury, T. Das, et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, 2012: 15–28.

[28] T. Parr and K. Fisher. LL(∗): the Foundation of the ANTLR Parser Generator. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, 2011: 425–436.

[29] T. Parr. The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2013.

[30] Toy Project. https://github.com/DonnyZone/ANTLR4-SqlBase.

[31] H. Li, A. Ghodsi, M. Zaharia, et al. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. Proceedings of the ACM Symposium on Cloud Computing, 2014: 6.

[32] Alluxio. https://www.alluxio.org/.

[33] 统一内存管理. http://www.linuxprobe.com/wp-content/uploads/2017/04/unified-memory-management-spark-10000.pdf.

[34] Sort Shuffle. https://issues.apache.org/jira/browse/SPARK-2045.

[35] Unsafe Shuffle. https://issues.apache.org/jira/browse/SPARK-7081.

[36] Consolidate Shuffle. https://issues.apache.org/jira/browse/SPARK-10708.

[37] Remove Hash Shuffle. https://issues.apache.org/jira/browse/SPARK-14667.

[38] Quadratic Probing. http://en.wikipedia.org/wiki/Quadratic_probing.

[39] Radix Sort. https://en.wikipedia.org/wiki/Radix_sort.

[40] TimSort. https://en.wikipedia.org/wiki/Timsort.

[41] Cache-aware Sort. https://issues.apache.org/jira/browse/SPARK-7079.

[42] C. Nyberg, T. Barclay, Z. Cvetanovic, et al. Alphasort: A Cache-Sensitive Parallel External Sort. VLDB J., 1995, 4(4): 603–627.

[43] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB, 2011, 4(9): 539–550.

[44] R. A. Lorie. XRM - An Extended (N-ary) Relational Memory. IBM Research Report, 1974: G320–2096.

[45] S. Padmanabhan, T. Malkemus, R. C. Agarwal, et al. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. Proceedings of the 17th International Conference on Data Engineering, 2001: 567–574.

[46] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-Pipelining Query Execution. CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, 2005: 225–237.

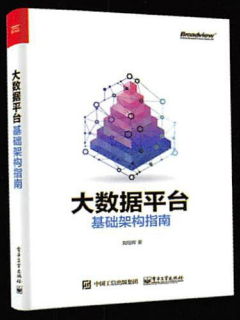
[47] M. Zukowski, P. A. Boncz, N. Nes, et al. Monetdb/x100 - A DBMS in the CPU cache. IEEE Data Eng. Bull., 2005, 28(2): 17–22.

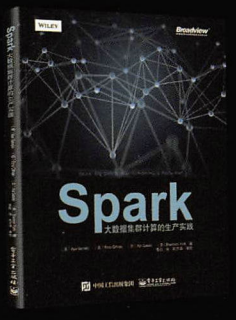
[48] J. Rao, H. Pirahesh, C. Mohan, et al. Compiled Query Execution Engine using JVM. Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 2006: 23.

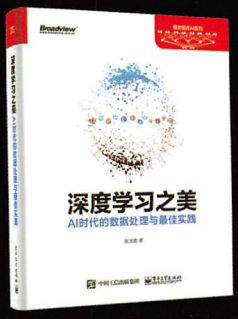
[49] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation, 2007.

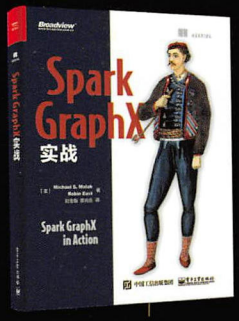
[50] 腾讯大数据平台. http://data.qq.com/.

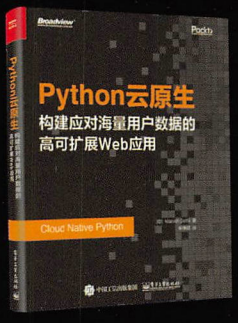
[51] History Server. https://issues.apache.org/jira/browse/SPARK-18085.


大数据平台
基础架构指南


Spark
大数据集群计算的生产实践


深度学习之美
AI时代的数据处理与最佳实践


Spark GraphX
实战
Spark GraphX in Action


Python云原生
构建应对海量用户数据的高可扩展Web应用
Cloud Native Python

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Spark SQL大大降低了数据分析人员使用Spark系统进行大数据处理的门槛。然而，相比网络社区的快速发展，相关的参考资料目前仍然比较匮乏，特别缺少对内部原理的深入解读。本书从源码层面给出了Spark SQL 从解析、优化到执行的全部图景，是难得的进阶图书。

周亮 百度高级工程师

本书作者长期从事分布式计算和查询优化方面的工作，对Spark SQL有较深入的研究及大量的工程积累。书中内容注重主线，从大量代码中提炼出纲要并按照模块展开，有点有面。读者在阅读过程中既能够从宏观上了然于胸，也能在特定的技术点深入其中。此外，作者分享的TDW实践经验对大数据平台建设和性能优化有着很重要的借鉴意义。

罗涛 小米人工智能与云平台研发工程师

Spark SQL是Spark生态圈中较活跃的部分，是近年来SQL-on-Hadoop解决方案中的佼佼者。本书深入地分析了Spark SQL的内部实现原理，更难能可贵的是，本书作者还分享了在大规模生产环境中的实践经验，干货非常多。我相信，无论是新手还是老手，都能够从中获益。

张呈刚 （www.linkedin.com/in/chenggangschool）AWS资深解决方案架构师

本书适合掌握了一些基础知识的读者阅读，如果你不满足于只知道Spark SQL是什么，而是想了解更多的实现原理，以及SQL这种声明式语言如何变为最终的RDD计算模型，那么本书将为你系统地答疑解惑。读者可以通过本书学习分布式系统和数据库实现等技术，理解系统背后的核心知识。

赵鑫 声智科技高级工程师

这是一本Spark高阶玩家的晋级宝典。作者深入浅出地剖析了Spark SQL的内部架构、算法设计与实现原理，让读者从知其然到知其所以然，从了解Spark、用好Spark到真正精通和改造Spark。

朱雷 （www.linkedin.com/in/lei-zhu-a4a63756）iGola 骑鹅旅行首席架构师

目前，同类书籍一般从Spark本身入手顺带涉及SQL模块的内容，但是在数据库的应用场景中，SQL模块往往是用户和开发人员较为关注的部分。本书正是以此为目标的进阶型图书，对于想对原生系统进行定制化改造的开发人员来说，这是一本实用的好书。

田斐 Oracle 工程师

本书结合前沿科研成果和一线工程实践，深入探讨了Spark SQL内部的原理，并分享了腾讯公司日均百万SQL总量和数百PB数据处理能力的平台建设经验。内容清晰而简明，讲解透彻而不枯燥，许多章节内容可提升读者触类旁通的能力。书中的背景概述等介绍有着学术论文的严谨，细节部分则适合结合源码参照理解。

纪树平 德国慕尼黑工业大学博士，Amazon软件工程师

上架建议：数据库

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：牛 勇
封面设计：侯士卿

ISBN 978-7-121-34314-8

9 787121 343148 >

定价：69.00元